

Static Analysis for Logic-Based Dynamic Programs*

Thomas Schwentick, Nils Vortmeier, and Thomas Zeume

TU Dortmund University
Germany

{thomas.schwentick, nils.vortmeier, thomas.zeume}@tu-dortmund.de

Abstract

A dynamic program, as introduced by Patnaik and Immerman (1994), maintains the result of a fixed query for an input database which is subject to tuple insertions and deletions. It can use an auxiliary database whose relations are updated via first-order formulas upon modifications of the input database.

This paper studies static analysis problems for dynamic programs and investigates, more specifically, the decidability of the following three questions. Is the answer relation of a given dynamic program always empty? Does a program actually maintain a query? Is the content of auxiliary relations independent of the modification sequence that lead to an input database? In general, all these problems can easily be seen to be undecidable for full first-order programs. Therefore the paper aims at pinpointing the exact decidability borderline for programs with restricted arity (of the input and/or auxiliary database) and restricted quantification.

1998 ACM Subject Classification F.4.1. Mathematical Logic

Keywords and phrases Dynamic descriptive complexity, algorithmic problems, emptiness, history independence, consistency

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

In modern database scenarios data is subject to frequent changes. In order to avoid costly re-computation of queries from scratch after each small modification of the data, one can try to use previously computed auxiliary data. This auxiliary data then needs to be updated dynamically whenever the database changes.

The descriptive dynamic complexity framework (short: dynamic complexity) by Patnaik and Immerman [21] models this setting from a declarative perspective. It was mainly inspired by updates in relational databases. Within this framework, for a relational database subject to change, a *dynamic program* maintains auxiliary relations with the intention to help answering a query Q . When a modification to the database, that is an insertion or deletion of a tuple, occurs, every auxiliary relation is updated through a first-order update formula (or, equivalently, through a core SQL query) that can refer to the database as well as to the auxiliary relations. The result of Q is, at every time, represented by some distinguished auxiliary relation. The class of all queries maintainable by dynamic programs with first-order update formulas is called DYNFO and we refer to such programs as DYNFO-programs. We note that shortly before the work of Patnaik and Immerman, the declarative approach was independently formalized in a similar way by Dong, Su and Topor [7].

* The first and third author acknowledge the financial support by DFG grant SCHW 678/6-1.



© Thomas Schwentick, Nils Vortmeier and Thomas Zeume;
licensed under Creative Commons License CC-BY

Conference title on which this volume is based on.

Editors: Billy Editor and Bill Editors; pp. 1–33



Leibniz International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The main question studied in Dynamic Complexity has been which queries that are not statically expressible in first-order logic (and therefore not in Core SQL), can be maintained by DYNFO-programs. Recently, it has been shown that the Reachability query, a very natural such query, can be maintained by DYNFO programs [2]. Altogether, research in Dynamic Complexity succeeded in proving that many non-FO queries are maintainable in DYNFO. These results and their underlying techniques yield many interesting insights into the the nature of Dynamic Complexity.

However, to complete the understanding of Dynamic Complexity, it would be desirable to complement these techniques by methods for proving that certain queries are *not* maintainable by DYNFO programs. But the state of the art with respect to inexpressibility results is much less favorable: at this point, no general techniques for showing that a query is not expressible in DYNFO are available. In order to get a better overall picture of Dynamic Complexity in general and to develop methods for inexpressibility proofs in particular, various restrictions of DYNFO have been studied, based on, e.g., arity restrictions for the auxiliary relations [3, 6, 4], fragments of first-order logic [14, 12, 26, 24], or by other means [5, 13].

At the heart of our difficulties to prove inexpressibility results in Dynamic Complexity is our limited understanding of what dynamic programs with or without restrictions “can do” in general, and our limited ability to analyze what a particular dynamic program at hand “does”. In this paper, we initiate a systematic study of the “analyzability” of dynamic programs. Static analysis of queries has a long tradition in Database Theory and we follow this tradition by first studying the emptiness problem for dynamic programs, that is the question, whether there exists an initial database and a modification sequence that is accepted by a given dynamic program.¹ Given the well-known undecidability of the finite satisfiability problem for first-order logic [22], it is not surprising that emptiness of DYNFO programs is undecidable in general. However, we try to pinpoint the borderline of undecidability for fragments of DYNFO based on restrictions of the arity of input relations, the arity of auxiliary relations and for the class DYNPROP of programs with quantifier-free update formulas.

In the fragments where undecidability of emptiness does not directly follow from undecidability of satisfiability in the corresponding fragment of first-order logic, our undecidability proofs make use of dynamic programs whose query answer might not only depend on the database yielded by a certain modification sequence, but also on the sequence itself, that is, on the order in which tuples are inserted or (even) deleted. From a useful dynamic program one would, of course, expect that it is *consistent* in the sense that its query answer always only depends on the current database, but not on the specific modification sequence by which it has been obtained. It turns out that the emptiness problem for consistent programs is easier than the general emptiness problem for dynamic programs. More precisely, there are fragments of DYNFO, for which an algorithm can decide emptiness for dynamic programs that come with a “consistency guarantee”, but for which the emptiness problem is undecidable, in general. However, it turns out that the combination of a consistency test with an emptiness test for consistent programs does not gain any advantage over “direct” emptiness tests, since the consistency problem turns out to be as difficult as the general emptiness problem.

Finally, we study a property that many dynamic programs in the literature share: they are *history independent* in the sense that all auxiliary relations always only depend on the

¹ The exact framework will be defined in Section 3, but we already mention that we will consider the setting in which databases are initially empty and the auxiliary relations are defined by first-order formulas.

	Emptiness Consistency	Emptiness for consistent programs	History Independence
Undecidable	DYNFO(1-in, 0-aux) DYNPROP(2-in, 0-aux) DYNPROP(1-in, 2-aux)	DYNFO(1-in, 2-aux) DYNFO(2-in, 0-aux)	DYNFO(2-in, 0-aux)
Decidable	DYNPROP(1-in, 1-aux)	DYNFO(1-in, 1-aux) DYNPROP(1-in) DYNPROP(1-aux)	DYNFO(1-in) DYNPROP(1-aux)
Open		DYNPROP(2-in, 2-aux) and beyond	DYNPROP(2-in, 2-aux) and beyond

■ **Table 1** Summary of the results of this paper. DYNFO(ℓ -in, m -aux) stands for DYNFO-programs with (at most) ℓ -ary input relations and m -ary auxiliary relations. DYNFO(m -aux) and DYNFO(ℓ -in) represent programs with m -ary auxiliary relations (and arbitrary input relations) and programs with ℓ -ary input relations, respectively. Likewise for DYNPROP.

current (input) database. History independence can be seen as a strong form of consistency in that it not only requires the query relation, but *all* auxiliary relations to be determined by the input database. History independent dynamic programs (also called *memoryless* [21] or *deterministic* [5]) are still expressive enough to maintain interesting queries like undirected reachability [13]. But also some inexpressibility proofs have been found for such programs [5, 13, 26]. We study the *history independence problem*, that is, whether a given dynamic program is history independent. In a nutshell, the history independence problem is the “easiest” of the static analysis problems considered in this paper.

Our results, summarized in Table 1, shed light on the borderline between decidable and undecidable fragments of DYNFO with respect to emptiness (and consistency), emptiness for consistent programs and history independence. While the picture is quite complete for the emptiness problem for general dynamic programs, for some fragments of DYNPROP there remain open questions regarding the emptiness problem for consistent dynamic programs and the history-independence problem. Some of the results shown in this paper have been already presented in the master thesis of Nils Vortmeier [23].

Outline We recall some basic definitions in Section 2 and introduce the formal setting in Section 3. The emptiness problem is defined and studied in Section 4, where we first consider general dynamic programs (Subsection 4.1) and then consistent dynamic programs (Subsection 4.2). In Subsection 4.3 we briefly discuss the impact of built-in orders to the results. The Consistency and History Independence problems are studied in Sections 5 and 6, respectively. We conclude in Section 7.

2 Preliminaries

We presume that the reader is familiar with basic notions from Finite Model Theory and refer to [10, 18] for a detailed introduction into this field. We review some basic definitions in order to fix notations.

In this paper, a *domain* is a non-empty finite set. For tuples $\vec{a} = (a_1, \dots, a_k)$ and $\vec{b} = (b_1, \dots, b_\ell)$ over some domain D , the $(k + \ell)$ -tuple obtained by concatenating \vec{a} and \vec{b} is denoted by (\vec{a}, \vec{b}) .

A (relational) *schema* is a collection τ of relation symbols² together with an arity function $\text{Ar} : \tau \rightarrow \mathbb{N}$. A *database* \mathcal{D} with schema τ and domain D is a mapping that assigns to every relation symbol $R \in \tau$ a relation of arity $\text{Ar}(R)$ over D . The *size of a database*, usually denoted by n , is the size of its domain. We call a database *empty*, if all its relations are empty. We emphasize that empty databases have non-empty domains. A τ -*structure* \mathcal{S} is a pair (D, \mathcal{D}) where \mathcal{D} is a database with schema τ and domain D . Often we omit the schema when it is clear from the context.

We write $\mathcal{S} \models \varphi(\vec{a})$ if the first-order formula $\varphi(\vec{x})$ holds in \mathcal{S} under the variable assignment that maps \vec{x} to \vec{a} . The *quantifier depth* of a first-order formula is the maximal nesting depth of quantifiers. The *rank- q type* of a tuple (a_1, \dots, a_m) with respect to a τ -structure \mathcal{S} is the set of all first-order formulas $\varphi(x_1, \dots, x_m)$ (with equality) of quantifier depth at most q , for which $\mathcal{S} \models \varphi(\vec{a})$ holds. By $\mathcal{S} \equiv_q \mathcal{S}'$ we denote that two structures \mathcal{S} and \mathcal{S}' have the same rank- q type (of length 0 tuples).

For a subschema $\tau' \subseteq \tau$, the rank- q τ' -type of a tuple \vec{a} in a τ -structure \mathcal{S} is its rank- q type in the τ' -reduct of \mathcal{S} .

We refer to the rank-0 type of a tuple also as its *atomic type* and, since we mostly deal with rank-0 types, simply as its *type*. The *equality type* of a tuple is the atomic type with respect to the empty schema.

The *k -ary type* of a tuple \vec{a} in a structure \mathcal{S} is its $\tau_{\leq k}$ -type, where $\tau_{\leq k}$ consists of all relation symbols of τ with arity at most k . The τ' -*color* of an element a in \mathcal{S} , for a subschema τ' of the schema of \mathcal{S} , is its τ'_1 -type, where τ'_1 consists of all unary relation symbols of τ' . We often enumerate the possible τ' -colors as c_0, \dots, c_L , for some L with c_0 being the color of elements that are in neither of the unary relations. We call these elements τ' -*uncolored*. If τ' is clear from the context we simply speak of colors and uncolored elements.

3 The dynamic complexity setting

For a database \mathcal{D} over schema τ , a *modification* $\delta = (o, \vec{a})$ consists of an operation $o \in \{\text{INS}_S, \text{DEL}_S \mid S \in \tau\}$ and a tuple \vec{a} of elements from the domain of \mathcal{D} . By $\delta(\mathcal{D})$ we denote the result of applying δ to \mathcal{D} with the obvious semantics of inserting or deleting the tuple \vec{a} to or from relation $S^{\mathcal{D}}$. For a sequence $\alpha = \delta_1 \cdots \delta_N$ of modifications to a database \mathcal{D} we let $\alpha(\mathcal{D}) \stackrel{\text{def}}{=} \delta_N(\cdots(\delta_1(\mathcal{D}))\cdots)$.

A *dynamic instance*³ of a query \mathcal{Q} is a pair (\mathcal{D}, α) , where \mathcal{D} is a database over a domain D and α is a sequence of modifications to \mathcal{D} . The dynamic query $\text{DYN}(\mathcal{Q})$ yields the result of evaluating the query \mathcal{Q} on $\alpha(\mathcal{D})$.

Dynamic programs, to be defined next, consist of an initialization mechanism and an update program. The former yields, for every (initial) database \mathcal{D} , an initial state with initial auxiliary data. The latter defines the new state of the dynamic program for each possible modification δ .

A *dynamic schema* is a pair $(\tau_{\text{in}}, \tau_{\text{aux}})$, where τ_{in} and τ_{aux} are the schemas of the input database and the auxiliary database, respectively. We call relations over τ_{in} *input relations* and relations over τ_{aux} *auxiliary relations*. If the relations are 0-ary, we also speak of input or auxiliary *bits*. We always let $\tau \stackrel{\text{def}}{=} \tau_{\text{in}} \cup \tau_{\text{aux}}$.

² For simplicity we do not allow constants in this work but note that our results hold for relational schemas with constants as well.

³ The following introduction to dynamic descriptive complexity is similar to previous work [26, 25].

► **Definition 3.1.** (Update program) An *update program* P over a dynamic schema $(\tau_{\text{in}}, \tau_{\text{aux}})$ is a set of first-order formulas (called *update formulas* in the following) that contains, for every $R \in \tau_{\text{aux}}$ and every $o \in \{\text{INS}_S, \text{DEL}_S \mid S \in \tau_{\text{in}}\}$, an update formula $\phi_o^R(\vec{x}; \vec{y})$ over the schema τ where \vec{x} and \vec{y} have the same arity as S and R , respectively.

A *program state* \mathcal{S} over dynamic schema $(\tau_{\text{in}}, \tau_{\text{aux}})$ is a structure $(D, \mathcal{I}, \mathcal{A})$ where⁴ D is a finite domain, \mathcal{I} is a database over the input schema (the *input database*) and \mathcal{A} is a database over the auxiliary schema (the *auxiliary database*).

The *semantics of update programs* is as follows. For a modification $\delta = (o, \vec{a})$, where \vec{a} is a tuple over D , and program state $\mathcal{S} = (D, \mathcal{I}, \mathcal{A})$ we denote by $P_\delta(\mathcal{S})$ the state $(D, \delta(\mathcal{I}), \mathcal{A}')$, where \mathcal{A}' consists of relations $R^{\mathcal{A}'} \stackrel{\text{def}}{=} \{\vec{b} \mid \mathcal{S} \models \phi_o^R(\vec{a}; \vec{b})\}$. The effect $P_\alpha(\mathcal{S})$ of a modification sequence $\alpha = \delta_1 \dots \delta_N$ to a state \mathcal{S} is the state $P_{\delta_N}(\dots(P_{\delta_1}(\mathcal{S}))\dots)$.

► **Definition 3.2.** (Dynamic program) A *dynamic program* is a triple (P, INIT, R_Q) , where

- P is an update program over some dynamic schema $(\tau_{\text{in}}, \tau_{\text{aux}})$,
- INIT is a mapping that maps τ_{in} -databases to τ_{aux} -databases, and
- $R_Q \in \tau_{\text{aux}}$ is a designated *query symbol*.

A dynamic program $\mathcal{P} = (P, \text{INIT}, R_Q)$ *maintains* a dynamic query $\text{DYN}(\mathcal{Q})$ if, for every dynamic instance (\mathcal{D}, α) , the query result $\mathcal{Q}(\alpha(\mathcal{D}))$ coincides with the query relation $R_Q^{\mathcal{S}}$ in the state $\mathcal{S} = P_\alpha(\mathcal{S}_{\text{INIT}}(\mathcal{D}))$, where $\mathcal{S}_{\text{INIT}}(\mathcal{D}) \stackrel{\text{def}}{=} (D, \mathcal{D}, \text{INIT}(\mathcal{D}))$ is the initial state for \mathcal{D} . If the query relation R_Q is 0-ary, we often denote this relation as *query bit* ACC and say that \mathcal{P} *accepts* α over D if ACC is true in $P_\alpha(\mathcal{S}_{\text{INIT}}(\mathcal{D}))$.

In the following, we write $\mathcal{P}_\alpha(\mathcal{D})$ instead of $P_\alpha(\mathcal{S}_{\text{INIT}}(\mathcal{D}))$ and $\mathcal{P}_\alpha(\mathcal{S})$ instead⁵ of $P_\alpha(\mathcal{S})$ for a given dynamic program $\mathcal{P} = (P, \text{INIT}, R_Q)$, a modification sequence α , an initial database \mathcal{D} and a state \mathcal{S} .

► **Definition 3.3.** (DYNFO and DYNPROP) DYNFO is the class of all dynamic queries that can be maintained by dynamic programs with first-order update formulas and first-order definable initialization mapping when starting from an initially empty input database. DYNPROP is the subclass of DYNFO, where update formulas are quantifier-free⁶.

A DYNFO-program is a dynamic program with first-order update formulas, likewise a DYNPROP-program is a dynamic program with quantifier-free update formulas. A DYNFO(ℓ -in, m -aux)-program is a DYNFO-program over (at most) ℓ -ary input databases that uses auxiliary relations of arity at most m ; likewise for DYNPROP(ℓ -in, m -aux)-programs.⁷

Due to the undecidability of finite satisfiability of first-order logic, the emptiness problem—the problem we study first—is undecidable even for DYNFO-programs with only a single auxiliary relation (more precisely, with query bit only). Therefore, we restrict our investigations to fragments of DYNFO. Also allowing arbitrary initialization mappings immediately yields an undecidable emptiness problem. This is already the case for first-order definable initialization mappings for arbitrary initial databases. In the literature classes with various restricted and unrestricted initialization mappings have been studied, see [25] for a discussion. In this work, in line with [21], we allow initialization mappings defined by arbitrary first-order formulas, but require that the initial database is empty. Of course, we could have studied

⁴ We prefer the notation $(D, \mathcal{I}, \mathcal{A})$ over $(D, \mathcal{I} \cup \mathcal{A})$ to emphasize the two components of the overall database.

⁵ The notational difference is tiny here: we refer to the dynamic program instead of the update program.

⁶ We still allow the use of quantifiers for the initialization.

⁷ We do not consider the case $\ell = 0$ where databases are pure sets with a fixed number of bits.

further restrictions on the power of the initialization formulas, but this would have yielded a setting with an additional parameter.

The following example illustrates a technique to maintain lists with quantifier-free dynamic programs, introduced in [12, Proposition 4.5], which is used in some of our proofs. The example itself is from [26].

► **Example 3.4.** We provide a DYNPROP-program \mathcal{P} for the dynamic variant of the Boolean query `NONEMPTYSET`, where, for a unary relation U subject to insertions and deletions of elements, one asks whether U is empty. Of course, this query is trivially expressible in first-order logic, but not without quantifiers.

The program \mathcal{P} is over auxiliary schema $\tau_{\text{aux}} = \{R_Q, \text{FIRST}, \text{LAST}, \text{LIST}\}$, where R_Q is the query bit (i.e. a 0-ary relation symbol), FIRST and LAST are unary relation symbols, and LIST is a binary relation symbol. The idea of \mathcal{P} is to maintain a list of all elements currently in U . The list structure is stored in the binary relation LIST^S . The first and last element of the list are stored in FIRST^S and LAST^S , respectively. We note that the order in which the elements of U are stored in the list depends on the order in which they are inserted into U .

For a given instance of `NONEMPTYSET` the initialization mapping initializes the auxiliary relations accordingly.

Insertion of a into U . A newly inserted element is attached to the end of the list⁸. Therefore the FIRST -relation does not change except when the first element is inserted into an empty set U . Furthermore, the inserted element is the new last element of the list and has a connection to the former last element. Finally, after inserting an element into U , the query result is 'true':

$$\begin{aligned}\phi_{\text{INS}}^{\text{FIRST}}(a; x) &\stackrel{\text{def}}{=} (\neg R_Q \wedge a = x) \vee (R_Q \wedge \text{FIRST}(x)) \\ \phi_{\text{INS}}^{\text{LAST}}(a; x) &\stackrel{\text{def}}{=} a = x \\ \phi_{\text{INS}}^{\text{LIST}}(a; x, y) &\stackrel{\text{def}}{=} \text{LIST}(x, y) \vee (\text{LAST}(x) \wedge a = y) \\ \phi_{\text{INS}}^{R_Q}(a) &\stackrel{\text{def}}{=} \top.\end{aligned}$$

Deletion of a from U . How a deleted element a is removed from the list, depends on whether a is the first element of the list, the last element of the list or some other element of the list. The query bit remains 'true', if a was not the first *and* last element of the list.

$$\begin{aligned}\phi_{\text{DEL}_U}^{\text{FIRST}}(a; x) &\stackrel{\text{def}}{=} (\text{FIRST}(x) \wedge x \neq a) \vee (\text{FIRST}(a) \wedge \text{LIST}(a, x)) \\ \phi_{\text{DEL}_U}^{\text{LAST}}(a; x) &\stackrel{\text{def}}{=} (\text{LAST}(x) \wedge x \neq a) \vee (\text{LAST}(a) \wedge \text{LIST}(x, a)) \\ \phi_{\text{DEL}_U}^{\text{LIST}}(a; x, y) &\stackrel{\text{def}}{=} x \neq a \wedge y \neq a \wedge (\text{LIST}(x, y) \vee (\text{LIST}(x, a) \wedge \text{LIST}(a, y))) \\ \phi_{\text{DEL}_U}^{R_Q}(a) &\stackrel{\text{def}}{=} \neg(\text{FIRST}(a) \wedge \text{LAST}(a))\end{aligned}$$

◀

In some parts of the paper we will use specific forms of modification sequences. An *insertion sequence* is a modification sequence $\alpha = \delta_1 \cdots \delta_m$ whose modifications are pairwise distinct insertions. An insertion sequence α over a unary input schema τ_{in} is in *normal form* if it fulfills the following two conditions.

- (N1) For each element a , the insertions affecting a form a contiguous subsequence α_a of α . We say that α_a *colors* a .
- (N2) For all elements a, b that get assigned the same τ_{in} -color by α , the projections of the subsequences α_a and α_b to their operations (i.e., their first parameters) are identical.

⁸ For simplicity we assume that only elements that are not already in U are inserted, the formulas given can be extended easily to the general case. Similar assumptions are made whenever necessary.

4 The Emptiness Problem

In this section we define and study the decidability of the emptiness problem for dynamic programs in general and for restricted classes of dynamic programs. The emptiness problem asks, whether the query relation R_Q of a given dynamic program \mathcal{P} is always empty, more precisely, whether $R_Q^S = \emptyset$ for every (empty) initial database \mathcal{D} and every modification sequence α with $S = \mathcal{P}_\alpha(\mathcal{D})$.

To enable a fine-grained analysis, we parameterize the emptiness problem by a class \mathcal{C} of dynamic programs.

Problem: EMPTINESS(\mathcal{C})

Input: A dynamic program $\mathcal{P} \in \mathcal{C}$ with FO initialization

Question: Is $R_Q^S = \emptyset$, for every initially empty database \mathcal{D} and every modification sequence α , where $S \stackrel{\text{def}}{=} \mathcal{P}_\alpha(\mathcal{D})$?

As mentioned before, undecidability of the emptiness problem for unrestricted dynamic programs follows immediately from the undecidability of finite satisfiability of first-order logic.

► **Theorem 4.1.** *EMPTINESS is undecidable for DYNFO(2-in, 0-aux)-programs.*

Proof. This follows easily from the undecidability of the finite satisfiability problem for first-order logic over schemas with at least one binary relation symbol [22]. For a given first-order formula φ over schema $\{E\}$ we construct a DYNFO-program \mathcal{P} with a single binary input relation E and a single 0-ary auxiliary relation ACC as follows. The bit ACC is set to true whenever the modified database is a model of φ , and set to false otherwise.

For correctness, we observe that if φ is not satisfiable then ACC is always false and therefore \mathcal{P} is empty. On the other hand, if φ is satisfiable, then there is a modification sequence α that is accepted by \mathcal{P} , so \mathcal{P} is non-empty. ◀

In the remainder of this section, we will shed some light on the border line between decidable and undecidable fragments of DYNFO. In Subsection 4.1 we study fragments of DYNFO obtained by disallowing quantification and/or restricting the arity of input and auxiliary relations. In Subsection 4.2, we consider dynamic programs that come with a certain consistency guarantee.

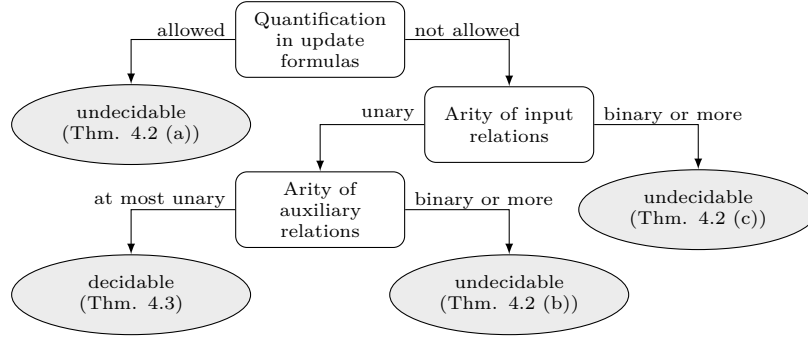
4.1 Emptiness of general dynamic programs

In this subsection we study the emptiness problem for various restricted classes of dynamic programs. We will see that the problem is basically only decidable if all relations are at most unary and no quantification in update formulas is allowed. Figure 1 summarizes the results.

At first we strengthen the general result from Theorem 4.1. We show that undecidability of the emptiness problem for DYNFO-programs holds even for unary input relations and auxiliary bits. Furthermore, quantification is not needed to yield undecidability: for DYNPROP-programs, emptiness is undecidable for binary input or auxiliary relations.

► **Theorem 4.2.** *The emptiness problem is undecidable for*

- (a) DYNFO(1-in, 0-aux)-programs,
- (b) DYNPROP(1-in, 2-aux)-programs,
- (c) DYNPROP(2-in, 0-aux)-programs,



■ **Figure 1** Decidability of EMPTINESS for various classes of dynamic programs.

Proof. In all three cases, the proof is by a reduction from the emptiness problem for semi-deterministic 2-counter automata.

In a nutshell, a counter automaton (short: CA) is a finite automaton that is equipped with counters that range over the non-negative integer numbers. A counter c can be incremented ($\text{inc}(c)$), decremented ($\text{dec}(c)$) and tested for zero ($\text{ifzero}(c)$). A CA does not read any input (i.e., its transitions can be considered to be ϵ -transitions) and in each step it can manipulate or test one counter and transit from one state to another state.

More formally, a CA is tuple (Q, C, Δ, q_i, F) , where Q is a set of states, $q_i \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and C is a finite set (the *counters*). The transition relation Δ is a subset of $Q \times \{\text{inc}(c), \text{dec}(c), \text{ifzero}(c) \mid c \in C\} \times Q$.

A *configuration* of a CA is a pair (p, \vec{n}) where p is a state and $\vec{n} \in \mathbb{N}^C$ gives a value n_c for each counter c in C . A transition $(p, \text{inc}(c), q)$ can be applied in state p , transits to state q and increments n_c by one. A transition $(p, \text{dec}(c), q)$ can be applied in state p if $n_c > 0$, transits to state q and decrements n_c by one. A transition $(p, \text{ifzero}(c), q)$ can be applied in state p , if $n_c = 0$ and transits to state q .

A *run* is a sequence of configurations consistent with Δ , starting from the *initial configuration* $(q_i, \vec{0})$. A run is *accepting*, if it ends in some configuration (q_f, \vec{n}) with $q_f \in F$. A CA is *deterministic* if Δ contains for every $p \in Q$ at most one transition (p, θ, q) . It is *semi-deterministic* if for every $p \in Q$ there is at most one transition (p, θ, q) in Δ or there are two transitions $(p, \text{dec}(c), q)$ and $(p, \text{ifzero}(c), q')$.

The emptiness problem for counter automata asks whether a given counter automaton has an accepting run. It follows from [20, Theorem 14.1-1] that the emptiness problem for semi-deterministic CA *with two counters* (2CA) is undecidable.⁹

In all three reductions, the dynamic program \mathcal{P} is constructed such that for every run ρ of the 2CA \mathcal{M} there is a modification sequence $\alpha = \alpha(\rho)$ that lets \mathcal{P} simulate ρ , and such that \mathcal{P} accepts on input α if and only if ρ is accepting. More precisely, the state of \mathcal{P} encodes the state of \mathcal{M} by auxiliary bits and the counters of \mathcal{M} in some way that differs in the three cases. However, in all cases it holds that not every modification sequence for \mathcal{P} corresponds to a run of \mathcal{M} . However, \mathcal{P} can detect if α does *not* correspond to a run and assume a rejecting sink state as soon as this happens.

For (a), the two counters are simply represented by two unary relations, such that the

⁹ The instruction set from [20] contains the increment instruction and a combined instruction that decrements a counter if it is non-zero and jumps to another instruction if it is zero. To simulate the latter instruction, we use two transitions $(p, \text{dec}(c), q)$ and $(p, \text{ifzero}(c), q')$ of which exactly one can be applied.

number of elements in a relation is the current value of the counter. The test whether a counter has value zero thus boils down to testing emptiness of a set and can easily be expressed by a formula with quantifiers.

The lack of quantifiers makes the reductions for (b) and (c) a bit more complicated. In both cases, the counters are represented by linked lists, where the number of elements in the list corresponds to the counter value (in (c): plus 1). With such a list a counter value zero can be detected without quantification. Due to the allowed relation types, the lists are built with auxiliary relations in (b) and with input relations in (c).

In the following, we describe more details of the reductions.

(a) We construct, from a semi-deterministic 2CA $\mathcal{M} = (Q, \{c_1, c_2\}, \Delta, q_I, F)$ a Boolean DYNFO(1-in, 0-aux)-program \mathcal{P} with unary input relations C_1 and C_2 and input bits Z_1 and Z_2 such that \mathcal{M} accepts a sequence θ of operations if and only if \mathcal{P} accepts a corresponding sequence α of modifications.

With a run ρ of \mathcal{M} we can associate an input sequence $\alpha(\rho)$ on a sufficiently large domain as follows: each transition of the form $(p, \text{inc}(c_i), q)$ gives rise to an insertion $\text{INS}_{C_i}(d)$, for some domain value d currently not in C_i . Likewise, each operation $(p, \text{dec}(c_i), q)$ corresponds to a deletion $\text{DEL}_{C_i}(d)$. Finally, operations $(p, \text{ifzero}(c_i), q)$ correspond alternatingly to operations $\text{INS}_{Z_i}()$ and $\text{DEL}_{Z_i}()$.

The semi-determinism of \mathcal{M} ensures that there is always at most one applicable transition and enables the program \mathcal{P} to keep track of the state of \mathcal{M} . The program ensures that only applicable transitions are taken.

The program \mathcal{P} has one auxiliary bit R_p for every state p of \mathcal{M} , an “error bit” R_e and the query bit ACC . During a “simulation” the current state p of \mathcal{M} corresponds to a program state in which exactly the auxiliary bit R_p is true (and ACC if $p \in F$). As soon as the input sequence contains an operation that does not correspond to an applicable transition of \mathcal{M} (either because no transition exists or because it can not be applied due to a counter value), the error bit R_e is switched on and remains on forever.

The update formulas of \mathcal{P} are as follows.

$$\begin{aligned}\phi_{\text{INS } C_i}^{R_q}(u) &\stackrel{\text{def}}{=} \neg C_i(u) \wedge \neg R_e \wedge \bigvee_{(p, \text{inc}(c_i), q) \in \Delta} R_p \\ \phi_{\text{INS } C_i}^{R_e}(u) &\stackrel{\text{def}}{=} R_e \vee C_i(u) \vee \bigvee_{p \in X} R_p \\ \phi_{\text{INS } C_i}^{\text{ACC}}(u) &\stackrel{\text{def}}{=} \neg C_i(u) \wedge \neg R_e \wedge \bigvee_{\substack{(p, \text{inc}(c_i), q) \in \Delta \\ \text{with } q \in F}} R_p\end{aligned}$$

Here, X is the set of states p from \mathcal{M} for which no transition $(p, \text{inc}(c_i), q)$ exists in Δ .

Deletions are handled similarly:

$$\begin{aligned}\phi_{\text{DEL } C_i}^{R_q}(u) &\stackrel{\text{def}}{=} C_i(u) \wedge \neg R_e \wedge \bigvee_{(p, \text{dec}(c_i), q) \in \Delta} R_p \\ \phi_{\text{DEL } C_i}^{R_e}(u) &\stackrel{\text{def}}{=} R_e \vee \neg C_i(u) \vee \bigvee_{p \in Y} R_p \\ \phi_{\text{DEL } C_i}^{\text{ACC}}(u) &\stackrel{\text{def}}{=} C_i(u) \wedge \neg R_e \wedge \bigvee_{\substack{(p, \text{dec}(c_i), q) \in \Delta \\ \text{with } q \in F}} R_p\end{aligned}$$

Here, Y is the set of states p from \mathcal{M} for which no transition $(p, \text{dec}(c_i), q)$ exists in Δ . Modifications to Z_i are handled as follows:

$$\begin{aligned}\phi_{\text{INS } Z_i}^{R_q}() &\stackrel{\text{def}}{=} \neg \exists x C_i(x) \wedge \neg R_e \wedge \bigvee_{(p, \text{ifzero}(c_i), q) \in \Delta} R_p \\ \phi_{\text{INS } Z_i}^{R_e}() &\stackrel{\text{def}}{=} R_e \vee \exists x C_i(x) \vee \bigvee_{p \in Z} R_p \\ \phi_{\text{INS } Z_i}^{\text{ACC}}() &\stackrel{\text{def}}{=} \neg \exists x C_i(x) \wedge \neg R_e \wedge \bigvee_{\substack{(p, \text{ifzero}(c_i), q) \in \Delta \\ \text{with } q \in F}} R_p\end{aligned}$$

Here, Z is the set of states p from \mathcal{M} for which no transition $(p, \text{ifzero}(c_i), q)$ exists in Δ . Deletions of input bits are handled exactly like insertions.

Now we prove that \mathcal{M} has an accepting run if and only if there is a modification sequence accepted by \mathcal{P} .

(only-if) Let ρ be an accepting run of \mathcal{M} and let m be the maximum value that a counter of \mathcal{M} assumes in ρ . It is not hard to prove by induction that there is a modification sequence on every domain with at least m elements that corresponds to ρ in the sense described above.

(if) For the other direction assume that $\alpha = \delta_1 \cdots \delta_n$ is a modification sequence over domain D that is accepted by \mathcal{P} . Let S_0 be the initial state of \mathcal{P} for D and let S_i for $i \in \{1, \dots, n\}$ be the state reached by \mathcal{P} after application of $\delta_1 \cdots \delta_i$. Then, by definition of the update formulas of \mathcal{P} and because S_n is accepting, the bit $R_e^{S_i}$ is not true for any S_i and no element is inserted into C_i when it was already contained in C_i , likewise elements are not deleted from C_i when they are not contained. The corresponding accepting run of \mathcal{M} is defined by the sequence $(q_0, \theta_0, q_1) \cdots (q_{n-1}, \theta_{n-1}, q_n)$ of transitions where q_i is the unique state q for which $R_q^{S_i}$ is true. Further the value for θ_i is $\text{inc}(c_j)$ if δ_{i+1} inserts an element into C_j , $\text{dec}(c_j)$ if δ_{i+1} deletes an element from C_j and $\text{ifzero}(c_j)$ if δ_{i+1} modifies Z_j .

(b) We note that in the proof of part (a) quantification is only needed for testing whether the input relations representing the counters are empty.

A DYNPROP(1-in, 2-aux)-program can simulate this check with two lists as in Example 3.4 for the relations C_1 and C_2 . When an insertion $\text{INS}_{C_i}(d)$ occurs, corresponding to an operation $(p, \text{inc}(c_i), q)$ in \mathcal{M} , the element d is appended to the end of the list for C_i . Analogously, for a deletion $\text{DEL}_{C_i}(d)$ the element d is removed from the list for C_i . As shown in Example 3.4 the dynamic program maintains auxiliary bits B_1, B_2 such that B_i is true if and only if C_i is not empty. These bits can then be used by the update formulas instead of the quantification. The rest of the proof is then analogous to the proof of (a).

(c) In this reduction the counters of the CA are represented by lists, as in (b), but the lists are encoded with (at most) binary *input relations*. Consequently, transitions of \mathcal{M} correspond to (bounded length) *sequences* of modifications for a dynamic program.

For each counter C_i the program \mathcal{P} use one binary input relation LIST_i , one unary input relation IN_i that contains all element used in the list, three unary input relations MIN_i , LAST_i , NEXTLAST_i to mark special elements, several auxiliary bits to monitor if all these input relations are used as intended and a bit NONEMPTY_i which states whether LIST_i is currently empty.

We now describe how to construct a modification sequence $\alpha = \alpha(\rho)$ from a run ρ of a given 2CA \mathcal{M} , that is accepted by \mathcal{P} if and only if ρ is accepting.

Before the actual simulation of \mathcal{M} can start, α has to initialize the input relations apart from LIST_i . To this end, \mathcal{P} expects as the first three modifications the insertion of one

element into MIN_i , LAST_i and IN_i . This element will serve as the head of the list.

A transition of \mathcal{M} that increments counter c_i is translated into a series of modifications that altogether insert a new element a into IN_i as follows. First, a is inserted into NEXTLAST_i and thus marked as to be inserted to the end of the list. Next the tuple (b, a) is inserted into LIST_i , where b is the unique element with $b \in \text{LAST}_i$. The list is surely not empty after the insertion of a , so NONEMPTY_i is set to true. After that, b is removed from LAST_i and a is inserted into IN , LAST_i and removed from NEXTLAST_i . If the modification sequence does not follow this protocol, \mathcal{P} assumes a rejecting state forever. Because every relation from MIN_i , LAST_i , NEXTLAST_i contains at most one element at every time, \mathcal{P} can indeed check whether all these modifications occur in the right order and on the right elements.

Similarly, a transition of \mathcal{M} decrementing c_i is translated into a series of modifications that altogether remove the unique element $a \in \text{LAST}_i$ from the corresponding list as follows. Let (b, a) be the tuple in LIST_i that contains a . The first modification has to be the insertion of b into NEXTLAST_i , after that (b, a) is deleted from LIST_i . If $b \in \text{MIN}_i$ then the list is now empty and NONEMPTY_i is set to false. a has to be removed from IN and LAST , b has to be inserted into LAST and removed from NEXTLAST .

It is straightforward but cumbersome to give the update formulas, so they are omitted here.

Otherwise, that is, besides the actual translation of a single step of \mathcal{M} , the proof is analogous to the proof of (a). ◀

The next result shows that emptiness of $\text{DYNPROP}(1\text{-in}, 1\text{-aux})$ -programs is decidable, yielding a clean boundary between decidable and undecidable fragments.

► **Theorem 4.3.** *EMPTINESS is decidable for $\text{DYNPROP}(1\text{-in}, 1\text{-aux})$ -programs.*

Proof. The proof uses the following two simple observations about $\text{DYNPROP}(1\text{-in}, 1\text{-aux})$ -programs \mathcal{P} .

- The initialization formulas of \mathcal{P} assign the same τ_{aux} -color to all elements. This color and the initial auxiliary bits only depend on the size of the domain. Furthermore there is a number $n(\mathcal{P})$, depending solely on the initialization formulas, such that the initial auxiliary bits and τ_{aux} -colors are the same for all empty databases with at least $n(\mathcal{P})$ elements. This observation actually also holds for $\text{DYNFO}(1\text{-in}, 1\text{-aux})$ -programs.
- When \mathcal{P} reacts to a modification $\delta = (o, a)$, the new (τ -)color of an element $b \neq a$ only depends on o , the old color of b , the old color of a , and the 0-ary relations. In particular, if two elements b_1, b_2 (different from a) have the same color before the update, they both have the same new color after the update. Thus, the overall update basically consists of assigning new colors to each color (for all elements except a), and the appropriate handling of the element a and the 0-ary relations.

We will show below that the behavior of $\text{DYNPROP}(1\text{-in}, 1\text{-aux})$ -programs can be simulated by an automaton model with a decidable emptiness problem, which we introduce next.

A *multicounter automaton* (short: MCA) is a counter automaton which is not allowed to test whether a counter is zero, i.e. the transition relation Δ is a subset of $Q \times \{\text{inc}(c), \text{dec}(c) \mid c \in C\} \times Q$. A *transfer multicounter automaton* (short: TMCA) is a multicounter counter automaton which has, in addition to the increment and the decrement operation, an operation that simultaneously transfers the content of each counter to another counter. More precisely the transition relation Δ is a subset of $Q \times (\{\text{inc}(c), \text{dec}(c) \mid c \in C\} \cup \{t \mid t : C \rightarrow C\}) \times Q$.

Applying a transition (p, t, q) to a configuration (p, \vec{n}) yields a configuration (q, \vec{n}') with $n'_c \stackrel{\text{def}}{=} \sum_{t(d)=c} n_d$ for every $c \in C$. A configuration (q, \vec{n}) of a TCMA is *accepting*, if $q \in F$. The emptiness problem for TCMA¹⁰ is decidable by reduction to the coverability problem for transfer petri nets¹¹ which is known to be decidable [9].

Let \mathcal{P} be a DYNPROP-program over unary schema $\tau = \tau_{\text{in}} \cup \tau_{\text{aux}}$ with query symbol R_Q which may be 0-ary or unary. Let Γ_0 be the set of all 0-ary (atomic) types over τ and let Γ_1 be the set of τ -colors. We construct a transfer multicounter automaton \mathcal{M} with counter set $Z_1 = \{z_\gamma \mid \gamma \in \Gamma_1\}$. The state set Q of \mathcal{M} contains Γ_0 , the only accepting state f and some further “intermediate” states to be specified below.

The intuition is that whenever \mathcal{P} can reach a state \mathcal{S} then \mathcal{M} can reach a configuration $c = (p, \vec{n})$ such that p reflects the 0-ary relations in \mathcal{S} and, for every $\gamma \in \Gamma_1$, n_γ is the number of elements of color γ in \mathcal{S} .

The automaton \mathcal{M} works in two phases. First, \mathcal{M} guesses the size n of the domain of the initial database. To this end, it increments the counter z_γ to n , where γ is the color assigned to all elements by the initialization formula for domains of size n , and it assumes the state corresponding to the initial 0-ary relations for a database of size n . Here the first of the above observations is used. Then \mathcal{M} simulates an actual computation of \mathcal{P} from the initial database of size n as follows. Every modification $\text{INS}_S(a)$ (or $\text{DEL}_S(a)$, respectively) in \mathcal{P} is simulated by a sequence of three transitions in \mathcal{M} :

- First, the counter z_γ , where γ is the color of a before the modification, is decremented.
- Second, the counters for all colors are adapted according to the update formulas of \mathcal{P} .
- Third, the counter $z_{\gamma'}$, where γ' is the color of a after the modification, is incremented.

If a modification changes an input bit, the first and third step are omitted. The state of \mathcal{M} is changed to reflect the changes of the 0-ary relations of \mathcal{P} . For this second phase the second of the above observations is used.

To detect when the simulation of \mathcal{P} reaches a state with non-empty query relation R_Q , states $p \in \Gamma_0$ may have a transition to the accepting state f .

Now we describe \mathcal{M} in detail. We begin with the simulation of the initialization step. If the quantifier depth of \mathcal{P} is q then \mathcal{M} non-deterministically guesses whether the domain is of size $1, \dots, q$ or at least $q + 1$. To this end the automaton has $q + 1$ additional states p_1, \dots, p_{q+1} , and non-deterministically chooses one such state p_i . Recall that the initial τ_{aux} -colors as well as the auxiliary bits depend only on the size of the domain, and that they are the same for all domains of size $\geq q + 1$. Let γ_0 be the 0-ary type and γ_1 be the color assigned to domains of size i . Now, \mathcal{M} increments the counter z_{γ_1} to i (or to at least i if $i = q + 1$) using some further intermediate states. Afterwards \mathcal{M} assumes state γ_0 .

Next we explain how a computation of \mathcal{P} is simulated. We first deal with modifications to unary input relations. As the effects of an update depend on the operation that is applied to an element, the color of that element and the 0-ary relations, \mathcal{M} has one chain of transitions for every such combination. So, for every state $p \in \Gamma_0$, every color $\gamma \in \Gamma_1$ and every $o \in \{\text{INS}_S, \text{DEL}_S\}$ with $S \in \tau_{\text{in}}$ and $\text{Ar}(S) = 1$ there are states $q_{p,\gamma,o}^1$ and $q_{p,\gamma,o}^2$ which are in charge of the simulation of an update when the modification $\delta = (o, a)$ occurs in a situation with 0-ary type p to an element a of color γ . A transition from p to $q_{p,\gamma,o}^1$ decreases the counter z_γ , a transition from $q_{p,\gamma,o}^2$ increases the counter for the new color of

¹⁰We note that (the complement of) this emptiness problem is often called *control-state reachability* problem.

¹¹The simulation of states by counters can be done as in [15, Lemma 2.1]

the modified element and assumes the state p' corresponding to the new 0-ary type. These two transitions simulate the changes of the auxiliary relations regarding the modified element. A transition from $q_{p,\gamma,o}^1$ to $q_{p,\gamma,o}^2$ handles the changes to the elements not (directly) affected by δ . As explained above, for given p , o and γ , the new color of an element depends only on its old color. From the update formulas of \mathcal{P} we extract a function $g_{p,\gamma,o} : \Gamma_1 \rightarrow \Gamma_1$ which describes these changes. From g we build the function $t : Z_1 \rightarrow Z_1$ that describes the transfer as $t(z_{\gamma'}) = z_{g_{p,\gamma,o}(\gamma')}$.

Similarly, modifications to input bits are simulated. Let $o \in \{\text{INS}_S, \text{DEL}_S\}$ with $S \in \tau_{\text{in}}$ and $\text{Ar}(S) = 0$ be an operation to a 0-ary input relation. For states $p, p' \in \Gamma_0$ there is a transition (p, t, p') if $t(z_{\gamma'}) = z_{g_{p,\gamma,o}(\gamma')}$ with $g_{p,\gamma,o} : \Gamma_1 \rightarrow \Gamma_1$ as above and p' corresponds to the 0-ary type after the update.

At last, transitions from $p \in \Gamma_0$ to f are introduced. The kind of these transitions depends on the arity of R_Q . If R_Q is 0-ary and $R_Q \in p$, then there is a transfer transition (p, id, f) where id is the identity. If R_Q is unary there is a transition $(p, \text{dec}(\gamma), f)$ for every color $\gamma \in \Gamma_1$ with $R_Q \in \gamma$.

It is not hard to show that there is a modification sequence for \mathcal{P} that leads to a non-empty query relation, if and only if there is a run of \mathcal{M} that reaches f . \blacktriangleleft

4.2 Emptiness of consistent dynamic programs

Some readers of the proof of Theorem 4.2 might have got the impression that we were cheating a bit, since the dynamic programs it constructs do not behave as one would expect: in all three cases each modification sequence α that yields a non-empty query relation R_Q can be changed, e.g., by switching two operations, into a sequence that does not correspond to a run of the CA and therefore does *not* yield a non-empty query relation. That is, the program \mathcal{P} is *inconsistent* because it might yield different results when the same database is reached through two different modification sequences.

It seems, that this inconsistency made the proof of Theorem 4.2 much easier. Therefore, the question arises, whether the emptiness problem becomes easier if it can be taken for granted that the given dynamic program is actually consistent. We study this question in this subsection and will investigate the related decision problem whether a given dynamic program is consistent in the next section.

As Table 1 shows, the emptiness problem for consistent dynamic programs is indeed easier in the sense that it is decidable for a considerably larger class of dynamic programs. While emptiness for general DYNFO programs is already undecidable for the tiny fragment with unary input relations and 0-ary auxiliary relations, it is decidable for consistent DYNFO programs with unary input and unary auxiliary relations. Likewise, for DYNPROP there is a significant gap: for consistent programs it is decidable for arbitrary input arities (with unary auxiliary relations) or arbitrary auxiliary arities (with unary input relations), but for general programs emptiness becomes undecidable as soon as binary relations are available (in the input *or* in the auxiliary database).

We call a dynamic program \mathcal{P} *consistent*, if it maintains a query with respect to an empty initial database, that is, if, for all modification sequences α to an empty initial database \mathcal{D}_\emptyset , the query relation in $\mathcal{P}_\alpha(\mathcal{D}_\emptyset)$ depends only on the database $\alpha(\mathcal{D}_\emptyset)$. In the remainder of this subsection we show the undecidability and decidability results stated in Table 1.

► **Theorem 4.4.** *The emptiness problem is undecidable for*

- (a) *consistent DYNFO(2-in, 0-aux)-programs, and*
- (b) *consistent DYNFO(1-in, 2-aux)-programs.*

Proof. Statement (a) is a corollary of the proof of Theorem 4.1, as the reduction in that proof always yields a consistent program.

For (b), we present another reduction from the emptiness problem for semi-deterministic 2CAs (see also the proof of Theorem 4.2). From a semi-deterministic 2CA \mathcal{M} we will construct a consistent Boolean dynamic program \mathcal{P} with a single unary input relation U . The query maintained by \mathcal{P} is “ \mathcal{M} halts after at most $|U|$ steps”. Clearly, such a program has a non-empty query result for some database and some modification sequence if and only if \mathcal{M} has an accepting run.

The general idea is that \mathcal{P} simulates one step of the run of \mathcal{M} whenever a new element is inserted to U . A slight complication arises from deletions from U , since it is not clear how one could simulate \mathcal{M} one step “backwards”. Therefore, when an element is deleted from U , \mathcal{P} freezes the simulation and stores the size m of $|U|$ before the deletion. It continues the simulation as soon as the current size ℓ of U grows larger than m , for the first time.

To help storing m and ℓ (and the values of the counters, for that matter), \mathcal{P} uses an auxiliary binary relation $R_<$ which, at any time, is a linear order on the set of those elements, that have been inserted to U at some point. Whenever an element is inserted to U for the first time, it becomes the maximum element of the linear order in $R_<$. Deletions and reinsertions do not affect $R_<$.

To actually store ℓ and m , \mathcal{P} uses two unary relations U_{current} and U_{max} . At any time, U_{current} contains the ℓ smallest elements with respect to $R_<$, where ℓ is the size of U at the time. Similarly, U_{max} contains the m smallest elements, with m as described above. In particular, U_{current} is empty if and only if $\ell = 0$. In the same fashion, \mathcal{P} uses two further unary auxiliary relations C_1 and C_2 representing the counters.

If \mathcal{M} reaches an accepting state, \mathcal{P} stores the current size k of U at this moment, with the help of another unary relation U_{acc} , that is, it simply lets U_{acc} become a copy of U_{current} after the current insertion. From that point on, that is, if U_{acc} is non-empty, the query bit of \mathcal{P} is true whenever $\ell \geq k$. Besides the one binary and five unary relations, \mathcal{P} has one 0-ary relation Q_p , for every state p of \mathcal{M} .

As an illustration we give two update formulas of \mathcal{P} that maintain C_1 and Q_q , for some state q , under insertions to U , respectively.

$$\begin{aligned}
\phi_{\text{INS } U}^{C_1}(u; x) &\stackrel{\text{def}}{=} ((U(u) \vee (U_{\text{current}} \neq U_{\text{max}})) \vee \bigvee_{\substack{(p, \text{inc}(c_2), q) \in \Delta \\ (p, \text{dec}(c_2), q) \in \Delta \\ (p, \text{ifzero}(c_2), q) \in \Delta}} Q_p) \wedge C_1(x)) \vee \\
&\quad \left(\neg U(u) \wedge (U_{\text{current}} = U_{\text{max}}) \wedge \right. \\
&\quad \left(\bigvee_{(p, \text{inc}(c_1), q) \in \Delta} (Q_p \wedge \forall y (C_1(y) \vee x \leq y)) \right. \\
&\quad \left. \vee \bigvee_{(p, \text{dec}(c_1), q) \in \Delta} (Q_p \wedge C_1(x) \wedge \exists y (C_1(y) \wedge x < y)) \right) \Big) \\
\phi_{\text{INS } U}^{Q_q}(u) &\stackrel{\text{def}}{=} ((U(u) \vee (U_{\text{current}} \neq U_{\text{max}})) \wedge Q_q) \vee \left(\neg U(u) \wedge (U_{\text{current}} = U_{\text{max}}) \wedge \right. \\
&\quad \left(\bigvee_{\substack{(p, \text{inc}(c_j), q) \in \Delta \\ j \in \{1, 2\}}} Q_p \right. \\
&\quad \left. \vee \bigvee_{\substack{(p, \text{dec}(c_j), q) \in \Delta \\ j \in \{1, 2\}}} (Q_p \wedge \exists x C_j(x)) \right) \Big)
\end{aligned}$$

$$\bigvee_{\substack{(p, \text{ifzero}(c_j), q) \in \Delta \\ j \in \{1, 2\}}} (Q_p \wedge \neg \exists x C_j(x))$$

Here, $U_{\text{current}} = U_{\text{max}}$ abbreviates the formula $\forall y (U_{\text{current}}(y) \leftrightarrow U_{\text{max}}(y))$. We note that $\phi_{\text{INS } U}^{C_1}$ does not test the applicability of transitions directly, but $\phi_{\text{INS } U}^{Q_q}$ does.

We recall that, thanks to semi-determinism of \mathcal{M} , the next transition is always uniquely determined by the state of \mathcal{M} and the value of the affected counter. If no transition can be applied, the simulation does not set any bit Q_i to true and the simulation basically stops. \blacktriangleleft

Contrary to the case of not necessarily consistent programs, the emptiness problem is decidable for consistent DYNFO(1-in, 1-aux)-programs. We will use the fact that the truth of first-order formulas with quantifier depth k in a state of a DYNFO(1-in, 1-aux)-program only depends on the number of elements of every color up to k .

Intuitively the states of a consistent DYNFO(1-in, 1-aux)-program can be approximated by a finite amount of information, namely the number of elements of every color up to some constant. This can be used to construct, from a consistent DYNFO(1-in, 1-aux)-program \mathcal{P} , a nondeterministic finite automaton \mathcal{A} that reads encoded modification sequences for \mathcal{P} in normal form and approximates the state of \mathcal{P} in its own state. In this way the emptiness problem for consistent DYNFO(1-in, 1-aux)-programs reduces to the emptiness problem for nondeterministic finite automata.

To formalize this, for a DYNFO(1-in, 1-aux)-program \mathcal{P} let c_1, \dots, c_M be the colors over the schema of \mathcal{P} . The *characteristic vector* $\vec{n}(\mathcal{S}) = (n_1, \dots, n_M)$ for a state \mathcal{S} over the schema of \mathcal{P} stores for every color c_i the number $n_i \in \mathbb{N}$ of elements of color c_i in \mathcal{S} . We also denote this number as $n_i(\mathcal{S})$. We write $n \simeq_k m$, for numbers k, n, m , if $n = m$ or both $n \geq k$ and $m \geq k$. We write $(n_1, \dots, n_M) \simeq_k (n'_1, \dots, n'_M)$, if for every $i \leq M$, $n_i \simeq_k n'_i$, and $\mathcal{S} \simeq_k \mathcal{S}'$ for two states \mathcal{S} and \mathcal{S}' if $\vec{n}(\mathcal{S}) \simeq_k \vec{n}(\mathcal{S}')$ and the bits in \mathcal{S} and \mathcal{S}' are equally valuated.

► **Lemma 4.5.** *Let \mathcal{P} be a DYNFO(1-in, 1-aux)-program with quantifier depth q and let \mathcal{S} and \mathcal{S}' be two states for \mathcal{P} .*

- (a) $\mathcal{S} \simeq_k \mathcal{S}'$ if and only if $\mathcal{S} \equiv_k \mathcal{S}'$ for any $k \in \mathbb{N}$.
- (b) Let a and a' be elements from \mathcal{S} and \mathcal{S}' with the same color c_i and let $k = q + 1$. If $\mathcal{S} \simeq_k \mathcal{S}'$ and $n_0(\mathcal{S}) \simeq_{k+1} n_0(\mathcal{S}')$ then $\mathcal{P}_{\delta(a)}(\mathcal{S}) \simeq_k \mathcal{P}_{\delta(a')}(\mathcal{S}')$ for every modification δ .

We recall that $\mathcal{S} \equiv_k \mathcal{S}'$ means that the two states satisfy exactly the same first-order formulas of quantifier depth (up to) k .

Proof. (a) It is easy to express with a first-order formula of quantifier depth k that the number of elements of a color c is exactly k' for $k' < k$ or at least k . So the only if direction follows. If $\mathcal{S} \simeq_k \mathcal{S}'$ holds, then Duplicator has a straightforward winning strategy in the k -rounds Ehrenfeucht-Fraïssé game, so $\mathcal{S} \equiv_k \mathcal{S}'$ follows.

(b) With part (a), $(\mathcal{S}, a) \equiv_k (\mathcal{S}', a')$. Since $k = q + 1$, if elements b and b' from \mathcal{S} and \mathcal{S}' have the same color and $b = a$ if and only if $b' = a'$, they also have the same color in $\mathcal{P}_{\delta(a)}(\mathcal{S})$ and $\mathcal{P}_{\delta(a')}(\mathcal{S}')$. The claim of the lemma follows. \blacktriangleleft

With the help of the previous lemma, we can now show the following decidability result.

► **Theorem 4.6.** *EMPTINESS is decidable for consistent DYNFO(1-in, 1-aux)-programs.*

Proof. We reduce the emptiness problem for consistent DYNFO(1-in, 1-aux)-programs to the emptiness problem for nondeterministic finite automata. The intuition is as follows. From a consistent DYNFO(1-in, 1-aux)-program \mathcal{P} , we construct a nondeterministic finite automaton \mathcal{A} that reads encoded modification sequences for \mathcal{P} in normal form and approximates the state of \mathcal{P} in its own state. To this end \mathcal{A} has a state $q_{\mathcal{E}}$ for every equivalence class \mathcal{E} of \simeq_k for a well-chosen $k \in \mathbb{N}$. The automaton accepts if it reaches a state $q_{\mathcal{E}}$ where \mathcal{E} corresponds to states of \mathcal{P} with non-empty query relation.

We make this more precise now. The following facts are exploited in the proof:

- As \mathcal{P} is consistent, if there is a modification sequence that leads to a state with a non-empty query relation, then there is an insertion sequence in normal form that leads to such a state.
- If two elements a, a' have the same color in some state of the program, then they still have the same color after an element $b \neq a, a'$ has been modified.
- For knowing how a state \mathcal{S} is updated by \mathcal{P} , it is enough to consider the \simeq_k equivalence class of \mathcal{S} for a suitable k .

In an insertion sequence in normal form, an element is touched by at most ℓ insertions where ℓ is the number of unary relation symbols in τ_{in} . As the insertions involving a single element occur consecutively in such a sequence, the occurring updates can be specified by “extended” update formulas of quantified depth ℓq , by nesting the original update formulas of quantifier depth q . For $k \stackrel{\text{def}}{=} \ell q + 1$, states \mathcal{S} and \mathcal{S}' with $\mathcal{S} \simeq_k \mathcal{S}'$ then meet the requirements of Lemma 4.5 (b) when those extended update formulas are considered.

The alphabet Σ of \mathcal{A} is the set of *proper* τ_{in} -colors ($\neq c_0$). For every equivalence class \mathcal{E} of \simeq_k , for k as chosen above, the automaton \mathcal{A} has a state $q_{\mathcal{E}}$. The idea is that the automaton simulates \mathcal{P} by approximating the state of \mathcal{P} by its \simeq_k -equivalence class. More precisely, whenever \mathcal{A} is in state $q_{\mathcal{E}}$ after reading a word w over Σ then \mathcal{E} is the equivalence class of the state \mathcal{S} reached by \mathcal{P} after the modification sequence α corresponding to w .

There is a small caveat to this. The state reached by \mathcal{P} after application of α is not solely determined by α but also by the size of the domain. The automaton has to take this into account.

We now describe the behaviour \mathcal{A} in detail. At the beginning of a computation the automaton non-deterministically guesses the (approximate) size of the domain, that is, a number i from $\{1, \dots, k\}$ and assumes state $q_{\mathcal{E}}$ where \mathcal{E} is the equivalence class of \simeq_k that corresponds to an initial state of \mathcal{P} with i elements if $i < k$ and at least i elements otherwise. Note that if $i = k$ then the automaton does not know the exact size of the domain for which it is simulating \mathcal{P} . Yet, as long as there are at least k τ_{in} -uncolored elements, the exact number is not important.

Afterwards \mathcal{A} simulates \mathcal{P} . When in state $q_{\mathcal{E}}$ and reading a symbol c , the automaton assumes state $q_{\mathcal{E}'}$ where \mathcal{E}' is as follows:

- If \mathcal{E} indicates less than k τ_{in} -uncolored elements then \mathcal{E}' is the \simeq_k -equivalence class of any state \mathcal{S}' reached by \mathcal{P} from a state \mathcal{S} with \simeq_k -equivalence class \mathcal{E} .
- If \mathcal{E} indicates at least k τ_{in} -uncolored elements, then \mathcal{A} guesses whether this is still the case after coloring one further element. If yes, then \mathcal{E}' is the \simeq_k -equivalence class of any state \mathcal{S}' reached by \mathcal{P} from a state \mathcal{S} with \simeq_k -equivalence class \mathcal{E} and at least $k + 1$ τ_{in} -uncolored elements. Otherwise \mathcal{E}' is the \simeq_k -equivalence class of any state \mathcal{S}' reached by \mathcal{P} from a state \mathcal{S} with \simeq_k -equivalence class \mathcal{E} and at least k τ_{in} -uncolored elements.

That \mathcal{E}' is uniquely determined follows from the second and third fact from above.

◀

The picture of decidability of emptiness for consistent programs for all classes of the form $\text{DYNFO}(\ell\text{-in}, m\text{-aux})$ is pretty clear and simple: it is decidable if and only if $\ell = 1$ and $m \leq 1$. Now we turn our focus to the corresponding classes of consistent DYNPROP -programs. Here we do not have a full picture. We show in the following that it is decidable if $\ell = 1$ or $m \leq 1$.

► **Theorem 4.7.** *The emptiness problem is decidable for*

- (a) *consistent $\text{DYNPROP}(1\text{-in})$ -programs.*
- (b) *consistent $\text{DYNPROP}(1\text{-aux})$ -programs.*

Proof (of Theorem 4.7 (a)). In [12, Theorem 3.2] it is shown that over databases with a linear order and unary relations every $\text{DYNPROP}(1\text{-in})$ -program \mathcal{P} with a Boolean query relation maintains a regular language over the τ_{in} -colors of the τ_{in} -colored elements. This result holds for arbitrary initialization and its proof shows that an automaton for this regular language can be effectively constructed from the dynamic program. Therefore, to test emptiness of a program with a Boolean query relation it suffices to test emptiness of its automaton.

Suppose that \mathcal{P} has a query relation with arity $k > 0$ and that there is a modification sequence α that yields a state \mathcal{S} where the query relation contains a tuple $\vec{a} \stackrel{\text{def}}{=} (a_1, \dots, a_k)$. Without loss of generality we assume that α is an insertion sequence in normal form and that elements of \vec{a} are modified at last (if they are modified at all). In other words, α is of the form $\alpha_1 \dots \alpha_M$ where each α_i modifies exactly one element, and there is an N such that α_j with $j \geq N$ only modifies elements of \vec{a} .

We use a pumping argument to argue that if α is a shortest such sequence, then it is not very long. Then emptiness of \mathcal{P} can be tested by examining all such modification sequences. We use the following observations from [12, Theorem 3.2]:

- (a) After each update, all tuples of positions that have not been touched so far have the same (atomic) type.
- (b) There is only a bounded number (depending only on the number and the maximal arity of the auxiliary relations of \mathcal{P}) of different types of such tuples.

Let \mathcal{S}_i be the state reached by applying $\alpha_1 \dots \alpha_i$. If N is larger than the number of (atomic) k -ary types then, by the observations (a) and (b), there are j, j' with $j < j'$ such that all k -tuples whose elements have not been touched so far have the same type in \mathcal{S}_j and $\mathcal{S}_{j'}$. In particular \vec{a} has the same type in \mathcal{S}_j and $\mathcal{S}_{j'}$. Hence, since \mathcal{P} is quantifier-free, it also has the same type in \mathcal{S} (the state reached by applying α) and in the state reached by applying the modification sequence $\alpha_1 \dots \alpha_j \alpha_{j'+1} \dots \alpha_N \alpha_{N+1} \dots \alpha_M$. Thus the query relation contains \vec{a} in the latter state. ◀

Before we prove the general statement of Theorem 4.7 (b), we first sketch the basic proof idea for consistent $\text{DYNPROP}(1\text{-aux})$ -programs over graphs, i.e., the input schema contains a single binary relation symbol E . For simplicity we also assume a 0-ary query relation. The general statement requires more machinery and is proved below.

Our goal is to show that if such a program \mathcal{P} accepts some graph then it also accepts one with “few” edges, where “few” only depends on the schema of the program. To this end we show that if a graph G accepted by \mathcal{P} contains many edges then one can find a large “well-behaved” edge set in G from which edges can be removed without changing the result of \mathcal{P} . Emptiness can then be tested in a brute-force manner by trying out insertion sequences for all graphs with few edges (over a canonical domain $\{1, \dots, n\}$).

More concretely, we consider an edge set “well-behaved”, if it consists only of self-loops, it is a set of disjoint non-self-loop-edges, or is a *star*, that is, the edges share the same source node or the same target node. From the Sunflower Lemma [11] it follows that for every $p \in \mathbb{N}$ there is an $N_p \in \mathbb{N}$ such that every (directed) graph with N_p edges contains p self-loops, or p disjoint edges, or a star with p edges.

Let us now assume, towards a contradiction, that the minimal graph accepted by \mathcal{P} has N edges with $N > N_{M^2+1}$, where M is the number of binary (atomic) types over the schema $\tau = \tau_{\text{in}} \cup \tau_{\text{aux}}$ of \mathcal{P} . Then G either contains $M^2 + 1$ self-loops, or $M^2 + 1$ disjoint edges, or a $(M^2 + 1)$ -star.

Let us assume first that G has a set $D \subseteq E$ of $M^2 + 1$ disjoint edges. We consider the state \mathcal{S} reached by \mathcal{P} after inserting all edges from $E \setminus D$ into the initially empty graph. Since D contains $M^2 + 1$ edges, there is a subset $D' \subseteq D$ of size $M + 1$ such that all edges in D' have the same atomic type in state \mathcal{S} . Let \mathcal{S}_0 be the state reached by \mathcal{P} after inserting all edges in $D \setminus D'$ in \mathcal{S} . All edges in D' still have the same type in \mathcal{S}_0 since \mathcal{P} is a quantifier-free program (though this type can differ from the type in \mathcal{S}). Let e_1, \dots, e_{M+1} be the edges in D' and denote by \mathcal{S}_i the state reached by \mathcal{P} after inserting e_1, \dots, e_i in \mathcal{S}_0 . For each i , all edges e_{i+1}, \dots, e_{M+1} have the same type γ_i in state \mathcal{S}_i , again. As the number of binary atomic types is M , there are $i < j$ such that $\gamma_i = \gamma_j$, thus e_{M+1} has the same type in \mathcal{S}_i and \mathcal{S}_j . Therefore, inserting the edges e_{j+1}, \dots, e_{M+1} in \mathcal{S}_i yields a state with the same query bit as inserting those edges in \mathcal{S}_j . As the query bit in the latter case is accepting, it is also accepting in the former case, yet in that case the underlying graph has fewer edges than G , the desired contradiction. The case where G contains $M^2 + 1$ self-loops is completely analogous.

Now assume that G contains a star with $M^2 + 1$ edges. The argument is very similar to the argument for disjoint edges. First insert all edges not involved in the star into an initially empty graph. Then there is a set D of many star edges of the same type, and they still have the same type after inserting the other edges of the star. A graph with fewer edges that is accepted by \mathcal{P} can then be obtained as above.

The idea generalizes to input schemata with larger arity by applying the Sunflower Lemma in order to obtain a “well-behaved” sub-relation within an input relation that contains many tuples. In order to prove this generalization we first recall the Sunflower Lemma, and observe that it has an analogon for tuples.

The Sunflower Lemma was introduced in [11], here we follow the presentation in [16]. A *sunflower* with p petals and a core Y is a collection of p sets S_1, \dots, S_p such that $S_i \cap S_j = Y$ for all $i \neq j$.

► **Lemma 4.8** (Sunflower Lemma, [11]). *Let $p \in \mathbb{N}$ and let \mathcal{F} be a family of sets each of cardinality ℓ . If \mathcal{F} consists of more than $N_{\ell,p} \stackrel{\text{def}}{=} \ell!(p-1)^\ell$ sets then \mathcal{F} contains a sunflower with p petals.*

We call a set H of tuples of some arity ℓ a *sunflower (of tuples)* if it has the following three properties.

- (i) All tuples in H have the same equality type.
- (ii) There is a set $J \subset \{1, \dots, \ell\}$ such that $t_j = t'_j$ for every $j \in J$ and all tuples $t, t' \in H$.
- (iii) For all tuples $t \neq t'$ in H the sets $\{t_i \mid i \notin J\}$ and $\{t'_i \mid i \notin J\}$ are disjoint.

We say that H has $|H|$ petals.

The following Sunflower Lemma for tuples has been stated in various variants in the literature, e.g., in [19, 17].

► **Lemma 4.9** (Sunflower Lemma for tuples). *Let $\ell, p \in \mathbb{N}$ and let R be a set of ℓ -tuples. If R contains more than $\bar{N}_{\ell,p} \stackrel{\text{def}}{=} \ell^\ell p^\ell (\ell!)^2$ tuples then it contains a sunflower with p petals.*

Proof. Let R be an ℓ -ary relation that contains $\bar{N}_{\ell,p}$ tuples. As there are less than ℓ^ℓ equality types of ℓ -tuples there is a set $R' \subseteq R$ of size at least $p^\ell (\ell!)^2$, in which all tuples have the same equality type. Application of Lemma 2 in [17] yields¹² a sunflower with p petals. ◀

It is instructive to see how Lemma 4.9 shows that a graph with sufficiently many edges has many selfloops, disjoint edges or a large star: Selfloops correspond to the equality type of tuples (t_1, t_2) with $t_1 = t_2$, many disjoint edges to the case $J = \emptyset$ and the two possible kinds of stars to $J = \{1\}$ and $J = \{2\}$, respectively.

Proof (of Theorem 4.7 (b)). Now the proof for binary input schemas easily translates to general input schemas. For the sake of completeness we give a full proof.

Suppose that a consistent DYNPROP(1-aux)-program \mathcal{P} over schema τ with 0-ary¹³ query relation accepts an input database \mathcal{D} that contains at least one relation R with many tuples.

Suppose that R is of arity ℓ and contains \bar{N}_{ℓ,M^2+1} diverse tuples where M is the number of ℓ -ary (atomic) types over the schema of \mathcal{P} . We show that \mathcal{P} already accepts a database with less tuples than \mathcal{D} .

By Lemma 4.9, R contains a sunflower R' of size $M^2 + 1$. Consider the state \mathcal{S} reached by \mathcal{P} after inserting all tuples from $R \setminus R'$ into the initially empty database. Since R' contains $M^2 + 1$ tuples, there is a subset $R'' \subseteq R'$ of size $M + 1$ such that all tuples in R'' have the same atomic type in state \mathcal{S} . Let \mathcal{S}_0 be the state reached by \mathcal{P} after inserting all tuples in $R' \setminus R''$ in \mathcal{S} . All tuples in R'' still have the same type in \mathcal{S}_0 since \mathcal{P} is a quantifier-free program (though this type can differ from the type in \mathcal{S}).

Let $\vec{a}_1, \dots, \vec{a}_{M+1}$ be the tuples in R'' and denote by \mathcal{S}_i the state reached by \mathcal{P} after inserting a_1, \dots, a_i in \mathcal{S}_0 . In state \mathcal{S}_i all tuples a_{i+1}, \dots, a_{M+1} have the same type, again. As the number of ℓ -ary atomic types is k , there are $i < j$ such that a_{M+1} has the same type in \mathcal{S}_i and \mathcal{S}_j . Therefore, inserting the edges e_{j+1}, \dots, e_{M+1} in \mathcal{S}_i yields a state with the same query bit as inserting this sequence in \mathcal{S}_j . As the query bit in the latter case is accepting, it is also accepting in the former case, yet in that case the underlying database has fewer tuples than \mathcal{D} , the desired contradiction.

If \mathcal{P} has a unary query relation, then the proof has to be adapted as follows. For an accepted database \mathcal{D} , the unary query relation contains some element a . Now M is chosen as the number of $(\ell + 1)$ -ary atomic types (instead of the number of ℓ -ary atomic types), and R'' is chosen as sub-sunflower where all tuples $(\vec{a}_1, a), \dots, (\vec{a}_{M+1}, a)$ have the same atomic type. The rest of the proof is analogous. ◀

The final result of this subsection gives a characterization of the class of queries maintainable by consistent DYNPROP(0-aux)-programs. This characterization is not needed to obtain decidability of the emptiness problem for such queries, since this is included in Theorem 4.7. However, we consider it interesting in its own right.

As DYNPROP(0-aux)-programs can only store a constant amount of information, it is not surprising that they can only maintain very simple properties. In fact, they can maintain exactly all modulo-like queries (to be defined precisely below). This characterization immediately yields an alternative emptiness test for consistent DYNPROP(0-aux)-programs.

¹² In [17], elements from the “outer part” of a petal can also occur in the “core”. As in R' all tuples have the same equality type, this can not happen in our setting.

¹³ At the end of the proof we discuss how to deal with unary query relations.

Furthermore it partially answers a question by Dong and Su [5]. They asked whether all queries maintainable by DYNFO(0-aux)-programs can already be maintained by history-independent DYNFO(0-aux)-programs. The characterization shows that this is the case for DYNPROP-programs, since all modulo-like queries can easily be maintained by history-independent DYNPROP(0-aux)-programs.

We first fix some notation. For a tuple $\vec{a} = (a_1, \dots, a_k)$ we write $\text{dom}(\vec{a})$ for the set $\{a_1, \dots, a_k\}$. The *cardinality* of \vec{a} is the size of $\text{dom}(\vec{a})$. The *strict underlying tuple* $\text{st}(\vec{a})$ is the tuple obtained from \vec{a} by removing all duplicate occurrences of data values (in a left-to-right fashion). A tuple \vec{a} is *duplicate-free* if $\text{st}(\vec{a}) = \vec{a}$.

A *strict atomic k-atom* is a relation atom $R(y_1, \dots, y_r)$ for which $\{y_1, \dots, y_r\} = \{x_1, \dots, x_k\}$ with $x_i \neq x_j$ for $i \neq j$. A *strict atomic k-type* $\gamma(x_1, \dots, x_k)$ is a set of strict atomic k -atoms. Let, for a tuple $\vec{a} = (a_1, \dots, a_k)$, ι be the valuation that maps, for each $j \in \{1, \dots, k\}$, x_j to a_j . Then the *strict atomic type* γ of tuple $\vec{a} = (a_1, \dots, a_k)$ in \mathcal{S} is the set of strict atomic k -atoms $R(y_1, \dots, y_r)$ in γ , for which $\iota(R(y_1, \dots, y_r))$ yields a fact in \mathcal{S} . We write $k\text{-type}(\vec{a})$ for the strict atomic type of a k -tuple \vec{a} .

However, the expressive power of consistent DYNPROP(0-aux)-programs can be most easily characterized in terms of types of sets of elements, rather than types of tuples.

The *set type* $\text{type}(A)$ of a set $A = \{a_1, \dots, a_k\}$ of size k in a structure \mathcal{S} is the set $\{k\text{-type}(\pi(\vec{a})) \mid \pi \in S_k\}$. Here, S_k denotes the set of permutations on $\{1, \dots, k\}$ and $\pi(\vec{a})$ denotes the tuple $(a_{\pi(1)}, \dots, a_{\pi(k)})$. We note that $\text{type}(A)$ does not depend on the chosen enumeration of A and is therefore well-defined. It directly follows from this definition that the set types of two sets with k elements are either equal or disjoint (as sets of strict atomic k -types). In other words, the strict atomic type of a set is determined by the strict atomic k -type of each duplicate-free tuple that can be constructed from elements of the set.

For a structure \mathcal{S} and a set type γ , we denote by $\#_{\mathcal{S}}(\gamma)$ the number of sets of set type γ in \mathcal{S} .

A *simple modulo expression* is an expression of the form $\#(\gamma) \equiv_p q$, where $p \geq 2$ and $q < p$ are natural numbers and γ is a non-empty set type. A structure \mathcal{S} satisfies such an expression if $\#_{\mathcal{S}}(\gamma) \equiv_p q$, that is, if the number of sets of type γ in \mathcal{S} has remainder q when divided by p . A *modulo expression* is a Boolean combination of simple modulo expressions. A *modulo query* is a query that can be defined as the set of all (finite) models of some modulo expression.

In the proof of the following theorem, we will consider modification sequences of a particular form that extends the normal form for insertion sequences over unary input schemas introduced in Section 3. A general insertion sequence α is in *normal form* if it fulfills the following three conditions.

- (M1) If α inserts tuples of cardinality k over a set A of k elements, then all such tuples are inserted in a contiguous subsequence α_A of α . Furthermore if α_A and $\alpha_{A'}$ are the contiguous sequences for sets A and A' with $|A| > |A'|$ then α_A occurs before $\alpha_{A'}$ in α .
- (M2) For all sets A, B with the same set type in \mathcal{I} , the subsequences α_A and α_B are isomorphic, that is, for some bijection $\pi : A \rightarrow B$, $\pi(\alpha_A) = \alpha_B$.

► **Theorem 4.10.** *A Boolean query \mathcal{Q} can be maintained by a consistent DYNPROP(0-aux) program if and only if it is a modulo query.*

Proof. (if) The set of Boolean queries that can be expressed by consistent DYNPROP(0-aux) programs is closed under all Boolean operators. It therefore suffices to show that each query defined by a simple modulo expression $\#(\gamma) \equiv_p q$ can be maintained by a consistent DYNPROP(0-aux) program \mathcal{P} .

The insertion of a tuple \vec{b} into some relation R changes the set type of exactly one set, $\{b_1, \dots, b_r\} \stackrel{\text{def}}{=} \text{dom}(\vec{b})$. It is straightforward but tedious to construct a quantifier-free formula $\varphi_\gamma^R(y_1, \dots, y_r)$ that expresses that the new type of the set $\{b_1, \dots, b_r\}$ after inserting \vec{b} to R is γ . Likewise, for the old set type of $\{b_1, \dots, b_r\}$. For deletions the situation is very similar. A DYNPROP(0-aux) program can therefore use p auxiliary bits to maintain the number of occurrences of set type γ in \mathcal{S} modulo p .

(only-if) Let \mathcal{P} be a consistent DYNPROP(0-aux)-program. As \mathcal{P} is consistent it yields, for each input database \mathcal{I} , the same query answer, for each modification sequence that results in \mathcal{I} . In this proof we therefore only consider insertion sequences in normal form.

Condition (M2) ensures that when a tuple \vec{b} is inserted to a relation R , there are no tuples present that involve a strict subset of $\text{dom}(\vec{b})$. As, on the other hand, due to the lack of quantifiers, the update formulas for the auxiliary bits can not take any tuples into account that contain elements outside of $\text{dom}(\vec{b})$, the auxiliary bits of \mathcal{P} after an insertion operation $\text{INS}_R(\vec{b})$ of α only depend on the current auxiliary bits of \mathcal{P} and the strict atomic k -type of $\text{st}(\vec{b})$. Similarly, by Condition (M3) it follows that the auxiliary bits after a modification subsequence α_A only depend on the current auxiliary bits of \mathcal{P} and the set type of A . The behavior of \mathcal{P} under a insertion sequence in normal form is therefore basically the behavior of a finite automaton (with the possible values of the auxiliary bits as states) reading a sequence of set types.¹⁴

Let m be the number of (0-ary) auxiliary bits of \mathcal{P} and let $M = (2^m)!$.

We next show that, for each non-empty set type γ and each two input databases \mathcal{I} and \mathcal{I}' that have for each non-empty set type different from γ the same number of sets and whose number of sets of type γ differs by M , either both \mathcal{I} and \mathcal{I}' are accepted by \mathcal{P} , or both are rejected. As there are only finitely many types and finitely many classes modulo M , this yields that the query decided by \mathcal{P} is a modulo query.

Let $\mathcal{S} = (D, \mathcal{I}, \mathcal{A})$ be some state reached after an insertion sequence α in normal form, let γ be some non-empty set type and let s be the number of occurrences of γ in \mathcal{I} . Let α' be the extension of α by $M + 2^m$ further sets of type γ yielding $\mathcal{S}' = (D, \mathcal{I}', \mathcal{A}')$. Let A_1, \dots, A_s denote the sets of type γ in \mathcal{I} and let $A_1, \dots, A_{s'}$ denote the sets of type γ in \mathcal{I}' . Let α' be decomposed into $\alpha_1 \alpha_{A_1} \dots \alpha_{A_s}, \alpha_2$.¹⁵ As there are only 2^m different possible values that the auxiliary bits can assume, there are $i < j, j \leq 2^m$, such that $\alpha_1 \alpha_{A_1} \dots \alpha_{A_i}$ and $\alpha_1 \alpha_{A_1} \dots \alpha_{A_j}$ yield states with identical auxiliary bits.¹⁶ As each set A_ℓ has the same set type, it follows that $\alpha_1 \alpha_{A_1} \dots \alpha_{A_{i+cd}}$ yields the same auxiliary bits as $\alpha_1 \alpha_{A_1} \dots \alpha_{A_i}$, for $d \stackrel{\text{def}}{=} j - i$ and every c with $i + cd \leq s + M + 2^m$. If $s \geq i$ it follows that $\alpha_1 \alpha_{A_1} \dots \alpha_{A_{i+M}}$ yields the same auxiliary bits as $\alpha_1 \alpha_{A_1} \dots \alpha_{A_i}$ and that $\alpha_1 \alpha_{A_1} \dots \alpha_{A_{s+M}}$ yields the same auxiliary bits as $\alpha_1 \alpha_{A_1} \dots \alpha_{A_s}$. Let us now assume that $s < i$. By deleting $i - s$ sets of type γ from the state reached after $\alpha_1 \alpha_{A_1} \dots \alpha_{A_i}$ and $\alpha_1 \alpha_{A_1} \dots \alpha_{A_{i+M}}$, we obtain states with identical auxiliary bits and s and $s + M$ sets of type γ , respectively. The claim then follows by adding back α_2 to the sequences $\alpha_1 \alpha_{A_1} \dots \alpha_{A_s}$ and $\alpha_1 \alpha_{A_1} \dots \alpha_{A_{s+M}}$, respectively. This completes the proof. \blacktriangleleft

¹⁴It should be noted here, that the overall number of set types is finite and only depends on the signature of \mathcal{P} .

¹⁵Note that α has the form $\alpha_1 \alpha_{A_1} \dots \alpha_{A_s} \alpha_2$.

¹⁶Here, $i = 0$ corresponds to the sequence α_1 .

4.3 The impact of built-in orders

A closer inspection of the proof that the emptiness problem is undecidable for consistent DYNFO(1-in, 2-aux)-programs (Theorem 4.4) reveals that the construction only requires one binary auxiliary relation: a linear order on the “active” elements. The proof would also work if a global linear order on all elements of the domain would be given. We say that a dynamic program has a *built-in linear order*, if there is one auxiliary relation $R_<$ that is always initialized by a linear order on the domain and never changed. Likewise, for a *built-in successor relation*.

That is, the border of undecidability for consistent DYNFO-programs actually lies between consistent DYNFO(1-in, 1-aux)-programs and consistent DYNFO(1-in, 1-aux)-programs with a built-in linear order. Similarly, the border of undecidability for (not necessarily consistent) DYNPROP-programs actually lies between DYNPROP(1-in, 1-aux)-programs and DYNPROP(1-in, 1-aux)-programs with a built-in linear order.

► **Proposition 4.11.** *The emptiness problem is undecidable for*

- (a) *consistent DYNFO(1-in, 1-aux)-programs with a built-in linear order or a built-in successor relation,*
- (b) *DYNPROP(1-in, 1-aux)-programs with a built-in successor relation.*

Proof. (a) The only binary auxiliary relation used in the proof of Theorem 4.4 was to simulate a linear order on the domain. This is not necessary any more, if the linear order is available. The linear order can easily be replaced by a built-in successor relation.

(b) We adapt the proof of Theorem 4.2 (b) and use the successor relation instead of the list relations, which are the only binary auxiliary relations. The first modification touches an element that is then marked as the first and last element of both lists. We then demand that an insertion to C_i inserts the element that is marked as last and a deletion from C_i deletes the predecessor of the last element. This can be checked and the marking of the last element can be updated without the use of quantifiers. A relation C_i is empty after the element that is marked as first is deleted from C_i .

◀

However, for dynamic programs that only have auxiliary bits, linear orders or successor relations do not affect decidability.

► **Proposition 4.12.** *The emptiness problem is decidable for*

- (a) *consistent DYNFO(1-in, 0-aux)-programs with a built-in linear order or a built-in successor relation,*
- (b) *DYNPROP(1-in, 0-aux)-programs with a built-in linear order or a built-in successor relation.*

Proof. (a) Let \mathcal{P} be a consistent program over unary input relations that uses only 0-ary auxiliary relations and a built-in linear order. In [8, Theorem 3.1] it is shown¹⁷ how to construct an existential monadic second order formula φ such that there is a modification sequence α with $\mathcal{P}_\alpha(\mathcal{D}_\emptyset)$ is accepted by \mathcal{P} if and only if $\alpha(\mathcal{D}_\emptyset) \models \varphi$. By [1], the formula φ describes a regular language over the proper τ_{in} -colors ($\neq c_0$). Hence an equivalent finite state automaton can be constructed. For finite automata the emptiness problem is decidable, so the claim follows.

¹⁷We note that the setting in that paper assumes a built-in linear order.

(b) This statement simply follows from the decidability of the emptiness problem for DYNPROP(1-in, 1-aux)-programs (Theorem 4.3) and the fact that the update formulas of DYNPROP(1-in, 0-aux)-programs only have one variable and therefore can not use a linear order or a successor relation in a non-trivial way. ◀

5 The Consistency Problem

In Section 4.2 we studied EMPTINESS for classes of consistent dynamic programs. It turned out that with this restriction the emptiness problem is easier than for general dynamic programs. One might thus consider the following approach for testing whether a given general dynamic program is empty: Test whether the program is consistent and if it is, use an algorithm for consistent programs. To understand whether this approach can be helpful, we study the following algorithmic problem, parameterized by a class \mathcal{C} of dynamic programs.

Problem: CONSISTENCY(\mathcal{C})

Input: A dynamic program $\mathcal{P} \in \mathcal{C}$ with FO initialization

Question: Is \mathcal{P} a consistent program with respect to empty initial databases?

We will see that the mentioned approach does not give us any advantage, as deciding CONSISTENCY is as hard as deciding EMPTINESS for general dynamic programs. It is not very surprising that CONSISTENCY is not easier than EMPTINESS, since deciding EMPTINESS boils down to finding *one* modification sequence resulting in a state with particular properties and CONSISTENCY is about finding *two* modification sequences resulting in two states with particular properties. This high level comparison can actually be turned into rather easy reductions from EMPTINESS to CONSISTENCY.

On the other hand, CONSISTENCY can also be reduced to EMPTINESS. For this direction the key idea is to simulate two modification sequences simultaneously and to integrate their resulting states into one joint state. This is easy if quantification is available, and requires some work for DYNPROP-fragments. We first give a technical lemma to restrict the kind of modification sequences that have to be considered to decide CONSISTENCY.

For this, we use the notion of *innocuous transformations*. Intuitively, an innocuous transformation θ of a modification sequence α is a minimal change of α that results in a modification sequence $\theta(\alpha)$ which leads to the same underlying database as α . Formally, an innocuous transformation is either (1) a permutation of a subsequence $\delta_1\delta_2$ to $\delta_2\delta_1$ under the condition that if one modification is $\text{INS}_S(\vec{a})$ then the other one is not $\text{DEL}_S(\vec{a})$, (2) the removal of a subsequence $\text{INS}_S(\vec{a})\text{DEL}_S(\vec{a})$ if \vec{a} is not contained in S when this subsequence is applied, (3) the removal of a modification $\delta = \text{INS}_S(\vec{a})$ or $\delta = \text{DEL}_S(\vec{a})$ if \vec{a} is already contained in S respectively \vec{a} is not contained in S when the modification is applied, or (4) the inverse of one of these transformations. It is clear that under the given conditions, for an innocuous transformation θ of a modification sequence α it holds that $\alpha(\mathcal{D}_\emptyset) = \theta(\alpha)(\mathcal{D}_\emptyset)$.

► **Lemma 5.1.** *Let \mathcal{P} be an inconsistent dynamic program. Then there is a modification sequence α , an innocuous transformation θ of α and an empty database \mathcal{D}_\emptyset such that the query relations in $\mathcal{P}_\alpha(\mathcal{D}_\emptyset)$ and $\mathcal{P}_{\theta(\alpha)}(\mathcal{D}_\emptyset)$ differ.*

Proof. As \mathcal{P} is inconsistent, there are two modification sequences α and α' that lead to the same input database \mathcal{I} but to states with different query relations. It is easy to see that $\alpha' = \theta_1 \cdots \theta_M(\alpha)$ where each θ_i is an innocuous transformation of $\theta_1 \cdots \theta_{i-1}(\alpha)$: From α and α' we can obtain a common insertion sequence α'' by applying innocuous transformations

of type (1)-(3), the inverses of the latter sequence of transformations then yields α' from α'' . As α and α' lead to states with different query relations there must be an i such that $\alpha^* \stackrel{\text{def}}{=} \theta_1 \cdots \theta_{i-1}(\alpha)$ and $\theta_i(\alpha^*)$ lead to states with different query relations. \blacktriangleleft

We now give the reductions between CONSISTENCY and EMPTINESS.

► **Theorem 5.2.** *Let $\ell \geq 1, m \geq 0$.*

- (a) For every $\mathcal{C} \in \{\text{DYNFO}(\ell\text{-in}, m\text{-aux}), \text{DYNFO}(\ell\text{-in}), \text{DYNFO}(m\text{-aux}), \text{DYNFO}\}$,
 - (i) $\text{EMPTINESS}(\mathcal{C}) \leq \text{CONSISTENCY}(\mathcal{C})$, and (ii) $\text{CONSISTENCY}(\mathcal{C}) \leq \text{EMPTINESS}(\mathcal{C})$.
- (b) For every $\mathcal{C} \in \{\text{DYNPROP}(\ell\text{-in}, m\text{-aux}), \text{DYNPROP}(\ell\text{-in}), \text{DYNPROP}(m\text{-aux}), \text{DYNPROP}\}$,
 - (i) $\text{EMPTINESS}(\mathcal{C}) \leq \text{CONSISTENCY}(\mathcal{C})$, and (ii) $\text{CONSISTENCY}(\mathcal{C}) \leq \text{EMPTINESS}(\mathcal{C})$.

Proof. For (a)(i) and (b)(i), we construct dynamic programs whose query relations are inflationary, that is, tuples that are inserted once are never removed afterwards. When an update adds a tuple and the modification that caused that update is undone, the two states that are reached after these updates are witnesses to inconsistency.

For (a)(ii) and (b)(ii), the constructed dynamic programs simulate two independent modification sequences and maintain two states of the original program. For (a)(ii), the program uses quantification to determine whether the two states represent equal input databases but different query relations. For (b)(ii) we use that thanks to Lemma 5.1 it suffices to simulate one modification sequence and at one point one innocuous transformation to find witnesses for inconsistency, so the two maintained states always represent equal input databases.

(a)(i) For a given $\text{DYNFO}(\ell\text{-in}, m\text{-aux})$ -program \mathcal{P} over schema $(\tau_{\text{in}}, \tau_{\text{aux}})$ with query symbol R_Q we construct a $\text{DYNFO}(\ell\text{-in}, m\text{-aux})$ -program \mathcal{P}' over $(\tau_{\text{in}}, \tau_{\text{aux}} \cup \{R'_Q\})$ with query symbol R'_Q . The idea is to initialize R'_Q as empty and add the tuples in R_Q to R'_Q with a delay of one modification. No tuple gets removed from R'_Q . The update formulas for R'_Q are $\phi_o^{R'_Q}(\vec{x}; \vec{y}) \stackrel{\text{def}}{=} R_Q(\vec{y}) \vee R'_Q(\vec{y})$. The update formulas for relations from τ_{aux} are copied from \mathcal{P} .

If \mathcal{P} is empty, then $R'_Q = \emptyset$ in every reached state \mathcal{S} and \mathcal{P}' is consistent. If \mathcal{P} is non-empty, then let α be a shortest modification sequence such that $R_Q^{\mathcal{P}_\alpha(\mathcal{D}_\emptyset)}$ is non-empty and let $\alpha^* = \alpha\alpha'$ be a modification sequence that leads to the same input database as α . It follows that the query relation R'_Q differs in $\mathcal{P}'_\alpha(\mathcal{D}_\emptyset)$ and $\mathcal{P}'_{\alpha^*}(\mathcal{D}_\emptyset)$ and \mathcal{P}' is inconsistent.

(a)(ii) If \mathcal{P} is a given $\text{DYNFO}(\ell\text{-in}, m\text{-aux})$ -program, we construct a $\text{DYNFO}(\ell\text{-in}, m\text{-aux})$ -program \mathcal{P}' that simulates two modification sequences of \mathcal{P} in parallel and maintains two states of this program. If the input databases of these states are equal, a tuple is added to the query relation of \mathcal{P}' if it is included in exactly one of the two maintained query relations of \mathcal{P} .

If \mathcal{P} is over schema $(\tau_{\text{in}}, \tau_{\text{aux}})$, then \mathcal{P}' is over schema $(\tau'_{\text{in}}, \tau'_{\text{aux}})$ where $\tau'_{\text{in}} = \{S, S' \mid S \in \tau_{\text{in}}\}$ and $\tau'_{\text{aux}} = \{R, R' \mid R \in \tau_{\text{aux}}\} \cup \{R_Q^*\}$. The query relation of \mathcal{P} is R_Q^* . The update formulas of relations $R \in \tau_{\text{aux}}$ are the same as in \mathcal{P} , for relations $R' \in \tau_{\text{aux}}$ the update formulas are obtained from the original formulas by replacing every relation symbol $S \in \tau_{\text{in}}$ or $R \in \tau_{\text{aux}}$ by S' or R' , respectively. The update formulas for R_Q^* first check if the two maintained input databases are equal by using conjunctions of formulas $\forall \vec{x}(S(\vec{x}) \leftrightarrow S'(\vec{x}))$ for every $S \in \tau_{\text{in}}$ and then inserts a tuple if it is in exactly one of the query relation R_Q of \mathcal{P} and its copy R'_Q . \mathcal{P} is consistent if and only if \mathcal{P}' is empty.

(b)(i) Analogous to (a)(i).

(b)(ii) We adapt the idea of part (a)(ii) with the help of Lemma 5.1. For a $\text{DYNPROP}(\ell\text{-in}, m\text{-aux})$ -program \mathcal{P} over schema $(\tau_{\text{in}}, \tau_{\text{aux}})$ we sketch the construction of a $\text{DYNPROP}(\ell\text{-in}, m\text{-aux})$ -program \mathcal{P}' over schema $(\tau'_{\text{in}}, \tau'_{\text{aux}})$. Like in part (a)(ii), this program simulates two modification sequences of \mathcal{P} and maintains two auxiliary databases over τ_{aux} , but only one input database over τ_{in} . Contrary to (a)(ii), \mathcal{P}' either simulates the effects of one modification to both auxiliary databases or, exactly once, a subsequence (of length at most 2) and an innocuous transformation of this subsequence. It follows that the input databases are equal for both simulated modification sequences after every simulated modification and so \mathcal{P}' only has to check whether there are tuples that are included in exactly one copy of the original query relation.

We now sketch the construction of \mathcal{P}' . Like in part (a)(ii), τ'_{aux} contains relation symbols R, R' for every $R \in \tau_{\text{aux}}$. Also all relation symbols from τ_{in} are contained in τ'_{in} . Additionally, τ'_{in} contains relation symbols U_S, I_S and T_S, T'_S for every $S \in \tau_{\text{in}}$ to simulate subsequences and their innocuous transformations. U_S is for simulating an unnecessary modification. If a modification $\text{INS}_{U_S}(\vec{a})$ is applied to \mathcal{P}' , the update formulas check that \vec{a} is already contained in S . If this check fails, \mathcal{P}' sets an error bit. Otherwise, \mathcal{P}' simulates \mathcal{P} for the modification $\text{INS}_S(\vec{a})$ on the second copy of the auxiliary database. Analogously for a modification $\text{DEL}_{U_S}(\vec{a})$. When a modification $\text{INS}_{I_S}(\vec{a})$ occurs, \mathcal{P}' simulates \mathcal{P} for the sequence $\text{INS}_S(\vec{a})\text{DEL}_S(\vec{a})$ on the second copy, if \vec{a} is not contained in S before. Otherwise, \mathcal{P}' sets an error bit. A sequence $\text{INS}_{T_S}(\vec{a})\text{INS}_{T'_S}(\vec{b})\text{DEL}_{T'_S}(\vec{b})\text{DEL}_{T_S}(\vec{a})$ is used to simulate the sequence $\text{INS}_S(\vec{a})\text{DEL}_S(\vec{b})$ on the first copy of the auxiliary database and the sequence $\text{DEL}_{S'}(\vec{b})\text{INS}_S(\vec{a})$ on the second copy, likewise for other combinations of insertions and deletions. Some additional auxiliary bits are used to check that four modifications like this happen in a row and that they do not represent the insertion of a tuple to a relation and the deletion of that tuple from the same relation. We use additional auxiliary bits to maintain whether exactly one innocuous transformation has been simulated. For every modification over relation symbols from τ_{in} , both copies of the auxiliary database get updated according to the original program \mathcal{P} .

It follows from Lemma 5.1 that it is possible for \mathcal{P}' to reach a state where the copies R_Q and R'_Q of the query relation of \mathcal{P} differ and no error bit is set if and only if \mathcal{P} is inconsistent. A tuple is inserted into the query relation R_Q^* of \mathcal{P}' when no error bit is set and the tuple is in exactly one of R_Q and R'_Q . So \mathcal{P}' is empty if and only if \mathcal{P} is consistent. ◀

6 The History Independence problem

As discussed in Section 4.2, it is natural to expect that a dynamic program is consistent, i.e., that the query relation only depends on the current database, but not on the modification sequence by which it has been reached. Many dynamic programs in the literature satisfy a stronger property: not only their query relation but *all* their auxiliary relations depend only on the current database. Formally, we call a dynamic program *history independent* if all auxiliary relations in $\mathcal{P}_\alpha(\mathcal{D})$ only depend on $\alpha(\mathcal{D})$, for all modification sequences α and initial empty databases \mathcal{D} . History independent dynamic programs (also called *memoryless* [21] or *deterministic* [5]) are still expressive enough to maintain interesting queries like undirected reachability [13], but also some lower bounds are known for such programs [5, 13, 26].

In this section, we study decidability of the question whether a given dynamic program is history independent.

Problem: HISTORYINDEPENDENCE(\mathcal{C})

Input: A dynamic program $\mathcal{P} \in \mathcal{C}$ with FO initialization

Question: Is \mathcal{P} history independent with respect to empty initial databases?

Note that contrary to the emptiness problem, HISTORYINDEPENDENCE is not easier for classes of consistent dynamic programs than for classes of general dynamic programs, so we will not study this restriction. This is because for every dynamic program \mathcal{P} we can construct a consistent dynamic program \mathcal{P}' that is history independent if and only if \mathcal{P} is, by introducing a new query bit that is not changed by any update formula.

Not surprisingly, HISTORYINDEPENDENCE is undecidable in general. This can be shown basically in the same way as the general undecidability of EMPTINESS in Theorem 4.1.

► **Theorem 6.1.** HISTORYINDEPENDENCE is undecidable for DYNFO(2-in, 0-aux)-programs.

Proof. Again we reduce from the satisfiability problem for first-order logic over schemas with at least one binary relation symbol. For a given FO-formula φ , at first we construct the dynamic program \mathcal{P} from the proof of Theorem 4.1. Additionally we add a second auxiliary bit B which is initialized as false and set to true when ACC is first set to true by an update, and never set to false again. If φ is not satisfiable, then all bits remain false and \mathcal{P} is history independent. If φ is satisfiable, then let $\alpha\delta$ be a shortest modification sequence applied to an empty database \mathcal{D}_\emptyset such that ACC and B are set to true in $\mathcal{P}_{\alpha\delta}(\mathcal{D}_\emptyset)$. Let δ^{-1} be the modification that undoes δ . Then B is false in $\mathcal{P}_\alpha(\mathcal{D}_\emptyset)$ and true in $\mathcal{P}_{\alpha\delta\delta^{-1}}(\mathcal{D}_\emptyset)$, but the respective input databases are equal. So \mathcal{P} is not history independent. ◀

However, in the following we will see that the precise borders between decidable and undecidable fragments are different for HISTORYINDEPENDENCE than for EMPTINESS and EMPTINESS for consistent programs. More precisely, we will show that HISTORYINDEPENDENCE is decidable for DYNFO- and DYNPROP-programs with unary input databases, and for DYNPROP-programs with unary auxiliary databases.

We recall the normal form for insertion sequences introduced in Section 3. For dynamic programs with unary input databases, insertion sequences in normal form (1) color each element contiguously and (2) apply the insertions for each τ_{in} -color in the same order. Here we require further that they first color all elements with designated τ_{in} -color c_1 , then all elements with c_2 and so on.

We will first show that to judge HISTORYINDEPENDENCE of DYNFO(1-in)-program only modification sequences in normal form (Lemma 6.2) and states with a particular property (Lemma 6.3) need to be considered. Finally, we show that if a dynamic program is not history independent, this can be observed already for domains of a bounded size in the size of the program (Proposition 6.7). The decision algorithm then tests all states over such “small” domains reached by insertion sequences in normal form in a brute-force manner.

Let \mathcal{P} be a DYNFO(1-in)-program over schema $\tau = \tau_{\text{in}} \cup \tau_{\text{aux}}$. Throughout this section we assume that τ contains only at least unary relation symbols and no input or auxiliary bits to ease presentation. This is no real restriction, as these bits can easily be simulated by unary relations when quantification is allowed. We usually denote the maximum quantifier depth of (initialization and update) formulas by q , the maximum arity of aux-relations by m , and the number of input relations by ℓ . Further we write L for $2^\ell - 1$ and let c_0, \dots, c_L be the τ_{in} -colors, where c_0 is the color of the τ_{in} -uncolored elements. In the following we write “colors” and “uncolored” instead of τ_{in} -colors and τ_{in} -uncolored.

We next present a characterization of history independence which is well-suited to

algorithmic analysis. We call a state \mathcal{S} over domain D *locally history independent*¹⁸ for a dynamic program \mathcal{P} if the following three conditions hold.

- (H1) $\mathcal{P}_{\delta_1 \delta_2}(\mathcal{S}) = \mathcal{P}_{\delta_2 \delta_1}(\mathcal{S})$ for all insertions δ_1 and δ_2 .
- (H2) $\mathcal{S} = \mathcal{P}_{\text{INS}_R(\vec{a})\text{DEL}_R(\vec{a})}(\mathcal{S})$ if $\vec{a} \notin R^{\mathcal{S}}$, for all $R \in \tau_{\text{in}}$ and \vec{a} over D .
- (H3) $\mathcal{S} = \mathcal{P}_{\text{INS}_R(\vec{a})}(\mathcal{S})$ if $\vec{a} \in R^{\mathcal{S}}$ and $\mathcal{S} = \mathcal{P}_{\text{DEL}_R(\vec{a})}(\mathcal{S})$ if $\vec{a} \notin R^{\mathcal{S}}$, for all $R \in \tau_{\text{in}}$ and \vec{a} over D .

► **Lemma 6.2.** *Let \mathcal{P} be a dynamic program.*

- (a) *\mathcal{P} is history independent if and only if every state reachable by \mathcal{P} via insertion sequences is locally history independent.*
- (b) *If \mathcal{P} is a DYNFO(1-in)-program, then \mathcal{P} is history independent if and only if every state reachable by \mathcal{P} via insertion sequences in normal form is locally history independent.*

Proof. (a) (only-if) It is easy to see that local history independence for all reachable states is necessary for history independence.

(if) Assume, towards a contradiction, that there is a dynamic program \mathcal{P} , for which every state reachable by an insertion sequence is locally history independent, but \mathcal{P} is not history independent. Then there are two modification sequences α_1 and α_2 to an empty database \mathcal{D}_\emptyset with $\alpha_1(\mathcal{D}_\emptyset) = \alpha_2(\mathcal{D}_\emptyset)$ but $\mathcal{P}_{\alpha_1}(\mathcal{D}_\emptyset) \neq \mathcal{P}_{\alpha_2}(\mathcal{D}_\emptyset)$. We construct insertion sequences α'_1 and α'_2 that lead to the same state as α_1 and α_2 , respectively. Repeated application of (H1) then yields $\mathcal{P}_{\alpha'_1}(\mathcal{D}_\emptyset) = \mathcal{P}_{\alpha'_2}(\mathcal{D}_\emptyset)$ and altogether $\mathcal{P}_{\alpha_1}(\mathcal{D}_\emptyset) = \mathcal{P}_{\alpha'_1}(\mathcal{D}_\emptyset) = \mathcal{P}_{\alpha'_2}(\mathcal{D}_\emptyset) = \mathcal{P}_{\alpha_2}(\mathcal{D}_\emptyset)$, the desired contradiction.

We only describe how to construct the insertion sequence α'_1 from α_1 ; the construction of α'_2 from α_2 is completely analogous. Let thus $\alpha_1 = \delta_1 \cdots \delta_N$ and, for every i , we denote by $\mathcal{S}_i \stackrel{\text{def}}{=} \mathcal{P}_{\delta_1 \cdots \delta_i}(\mathcal{D}_\emptyset)$.

A modification is *bad* if it is a deletion or the repeated insertion of a fact. The insertion sequence α'_1 is constructed by successively eliminating all bad modifications from α_1 . If α_1 does not contain any bad modification, we are done. Otherwise, let δ_k be the first bad modification in α_1 . Since $\delta_1 \cdots \delta_{k-1}$ is an insertion sequence, by our assumption \mathcal{S}_{k-1} is locally history independent. Therefore, δ_k can be eliminated from α_1 as follows. If $\delta_k = \text{DEL}_R(\vec{a})$ such that $\vec{a} \notin R^{\mathcal{S}_{k-1}}$ or $\delta_k = \text{INS}_R(\vec{a})$ such that $\vec{a} \in R^{\mathcal{S}_{k-1}}$ then $\mathcal{S}_k = \mathcal{S}_{k-1}$ thanks to (H3) and δ_k can be removed from α_1 without affecting the resulting state. If $\delta_k = \text{DEL}_R(\vec{a})$ such that $\vec{a} \in R^{\mathcal{S}_{k-1}}$, then there must be an insertion $\text{INS}_R(\vec{a})$ in $\delta_1 \cdots \delta_{k-1}$. By (H1) the insertions $\delta_1 \cdots \delta_{k-1}$ can be rearranged into a sequence $\beta \text{INS}_R(\vec{a})$, such that β consists of all modifications from $\delta_1 \cdots \delta_{k-1}$ besides $\text{INS}_R(\vec{a})$ and the resulting state is \mathcal{S}_{k-1} . By (H2), the modification sequences β and $\beta \text{INS}_R(\vec{a}) \text{DEL}_R(\vec{a})$ yield the same state, but β has fewer deletions than $\delta_1 \cdots \delta_k$. The modification sequence α'_1 is obtained by repeating this procedure.

(b) (only-if) Again, local history independence for all reachable states is necessary for history independence.

(if) Let \mathcal{P} be a dynamic program for which every state reachable via a insertion sequence in normal form is locally history independent. We show that every state reachable by an insertion sequence is also reachable by a normal form sequence. That \mathcal{P} is history independent then follows from (a).

We thus assume, towards a contradiction, that there is an insertion sequence $\alpha = \delta_1 \cdots \delta_N$ and an empty database \mathcal{D}_\emptyset such that $\mathcal{S} = \mathcal{P}_\alpha(\mathcal{D}_\emptyset)$ is not reachable by any insertion sequence in

¹⁸ We define this term for arbitrary input arity, since the first part of Lemma 6.2 holds in general.

normal form. Let α and \mathcal{D}_\emptyset be chosen such that N is minimal. Therefore, $\mathcal{S}' = \mathcal{P}_{\delta_1 \dots \delta_{N-1}}(\mathcal{D}_\emptyset)$ can be reached by a normal form modification¹⁹ sequence $\alpha' = \delta'_1 \dots \delta'_{N-1}$ and, by our assumption, \mathcal{S}' and all prior states reached by prefixes of α' are locally history independent. By inductive application of (H1), δ_N can now be moved to its appropriate place inside α' to yield a normal form sequence α'' equivalent to α . Therefore, \mathcal{S} is reachable by a normal form sequence, the desired contradiction. \blacktriangleleft

We next define another property that reachable states of history independent programs share. A state \mathcal{S} is *homogeneous* if all tuples \vec{a} and \vec{b} with the same (atomic) τ_{in} -type also have the same (atomic) τ_{aux} -type. For every homogeneous state \mathcal{S} we denote by $f_{\mathcal{S}}$ the (atomic) *type function* that maps every (atomic) τ_{in} -type of arity m (the maximal arity of τ) to the corresponding (atomic) τ_{aux} -type²⁰. The following lemma is an immediate consequence of [5, Lemma 16].

► **Lemma 6.3.** *For every history independent DYNFO(1-in)-program, every reachable state is homogeneous.*

We call a state of a DYNFO(1-in)-program that is not homogeneous or not locally history independent a *bad state*. That a state is bad can be expressed in first-order logic. Likewise the possible effects of coloring a single uncolored element on the type function of a state can be expressed by first-order formulas. To state this more precisely, we use *type forecast functions* $F : \{1, \dots, L\} \rightarrow \mathcal{F}$, where \mathcal{F} is the set of possible type functions for \mathcal{P} .

► **Lemma 6.4.** *Let \mathcal{P} be a DYNFO(1-in, m -aux)-program with maximum quantifier-depth q and ℓ input relations.*

- (a) *There is a formula φ_{bad} of quantifier-depth at most $3 + 2m + (\ell + 1)q$ that is true in a state \mathcal{S} if and only if $\mathcal{P}_\alpha(\mathcal{S})$ is bad for at least one modification sequence α that colors a single uncolored element of \mathcal{S} .*
- (b) *For every type forecast function F there is a formula φ_F of quantifier depth $1 + m + \ell q$ that is true in a homogeneous state \mathcal{S} if and only if, for every $i \leq L$, $\mathcal{P}_\alpha(\mathcal{S})$ has type function $F(i)$ if α colors some uncolored element with c_i .*

Proof. (a) The formula is of the form

$$\exists x \bigvee_{i=1}^L (\varphi_1^i \vee \varphi_2^i),$$

where φ_1^i expresses that the state that results from coloring an uncolored element by c_i is not homogeneous and φ_2^i expresses that it is not locally history independent.

To this end, φ_1^i existentially quantifies two m -tuples (depth: $2m$) and expresses that they have the same τ_{in} -type but different τ_{aux} -types in the state after the coloring (depth: ℓq).

The formula φ_2^i is a three-fold disjunction for the conditions (H1-3). As an example, the formula for (H1) quantifies two elements a, a' (depth: 2), an m -tuple (depth: m) and tests that for some color c_i the τ_{aux} -types of the two databases resulting from the two possible orders in which a and a' can be colored by c_i (depth: $2q$) differ in the m -tuple.

Altogether, φ_{bad} has quantifier-depth $1 + \max(2m + \ell q, 2 + m + 2q) \leq 3 + 2m + (\ell + 1)q$.

¹⁹ Of course, insertion sequences yielding the same state have the same length.

²⁰ If there is no tuple \vec{a} of an τ_{in} -type c in \mathcal{S} , then $f_{\mathcal{S}}(c) = \perp$.

(b) Similarly, each formula φ_F existentially quantifies an element a to be colored, has a disjunct for all possible colors, and universally quantifies an m -tuple and tests that the τ_{aux} -type of it is consistent with its τ_{in} -type and F . Overall this yields quantifier depth $1 + m + \ell q$.

◀

We next formalize the observation that for a homogeneous state, the truth of first-order formulas of quantifier depth k only depends on its color frequencies up to k^{21} . To this end, we associate with every state \mathcal{S} its *characteristic input vector* $\vec{n}^{\text{in}}(\mathcal{S}) = (n_0, \dots, n_L)$ over \mathbb{N} where $n_i \stackrel{\text{def}}{=} n_i^{\text{in}}(\mathcal{S})$ is the number of elements with τ_{in} -color c_i in \mathcal{S} .

We write $n \simeq_k m$, for numbers k, n, m , if $n = m$ or both $n \geq k$ and $m \geq k$. We write $(n_0, \dots, n_L) \simeq_k (n'_0, \dots, n'_L)$, if for every $i \leq L$, $n_i \simeq_k n'_i$.

For a given k , we say that two homogeneous states \mathcal{S} and \mathcal{S}' are k -similar (denoted by $\mathcal{S} \sim_k \mathcal{S}'$) if

- $\vec{n}^{\text{in}}(\mathcal{S}) \simeq_k \vec{n}^{\text{in}}(\mathcal{S}')$ and
- \mathcal{S} and \mathcal{S}' have the same type function.

Now we can make the relationship between characteristic input vectors and first-order types more precise.²²

► **Lemma 6.5.** *Let \mathcal{P} be a DYNFO(1-in, m -aux)-program and let \mathcal{S} and \mathcal{S}' be two homogeneous states for \mathcal{P} . For every $k \in \mathbb{N}$, if $\mathcal{S} \sim_k \mathcal{S}'$ then $\mathcal{S} \equiv_k \mathcal{S}'$.*

We recall that $\mathcal{S} \equiv_k \mathcal{S}'$ means that the two states satisfy exactly the same first-order formulas of quantifier depth (up to) k .

Proof. If $\mathcal{S} \sim_k \mathcal{S}'$ then the duplicator has a straightforward winning strategy for the k -round Ehrenfeucht-Fraïssé game on the τ_{in} -reducts of \mathcal{S} and \mathcal{S}' . Since both states are homogeneous and have the same type function, this winning strategy extends to τ_{aux} and the strategy of duplicator is a winning strategy for \mathcal{S} and \mathcal{S}' . ◀

By combining Lemmas 6.4 and 6.5 we get the following lemma, which will be the most important technical tool in the proof of a small counterexample property for programs that are not history independent.

► **Lemma 6.6.** *Let \mathcal{P} be a DYNFO(1-in, m -aux)-program with maximum quantifier-depth q and ℓ input relations, let $K \geq 1 + m + \ell q$ and let \mathcal{S} and \mathcal{S}' be two homogeneous states for \mathcal{P} with $\mathcal{S} \sim_K \mathcal{S}'$. Let a and a' be uncolored elements in \mathcal{S} and \mathcal{S}' , respectively. Let β and β' be insertion sequences that color a and a' , respectively with the same color c_i . Then $\mathcal{P}_\beta(\mathcal{S})$ and $\mathcal{P}_{\beta'}(\mathcal{S}')$ have the same type function, in case they are both homogeneous.*

Proof. By Lemma 6.5, we know that $\mathcal{S} \equiv_K \mathcal{S}'$. In particular, thanks to Lemma 6.4 and the homogeneity of $\mathcal{P}_\beta(\mathcal{S})$ and $\mathcal{P}_{\beta'}(\mathcal{S}')$, there is a unique type forecast function F such that φ_F holds in \mathcal{S} and \mathcal{S}' . Therefore, after coloring a and a' with c_i the resulting states both have type function $F(i)$. ◀

Now we can show a small counterexample property for programs that are not history independent.

²¹ Note the similarities to Lemma 4.5

²² We note that for homogeneous states it actually holds: $\mathcal{S} \sim_k \mathcal{S}'$ if and only if $\mathcal{S} \equiv_k \mathcal{S}'$.

► **Proposition 6.7.** *Let \mathcal{P} be a DYNFO(1-in, m -aux)-program with quantifier depth q and ℓ input relations, and let $K \stackrel{\text{def}}{=} 3 + 2m + (\ell + 1)q$ and T be the number of type functions. If \mathcal{P} is not history independent, then there exists a database \mathcal{D}_0 of size at most $N \stackrel{\text{def}}{=} (2K + T)(L + 1)$ and a insertion sequence in normal form α such that $\mathcal{P}_\alpha(\mathcal{D}_0)$ is bad.*

Proof. Let \mathcal{P} be a dynamic DYNFO(1-in, m -aux)-program that is not history independent and let \mathcal{D}_0 be an empty database of minimal size n for which there exists a insertion sequence in normal form $\alpha_1 \cdots \alpha_N$, such that $\mathcal{P}_\alpha(\mathcal{D}_0)$ is bad, each subsequence α_i colors one element, and N is minimal.

We consider the state $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{P}_{\alpha_1 \cdots \alpha_{N-1}}(\mathcal{D}_0)$ just before the bad state. Thus \mathcal{S} satisfies the formula φ_{bad} from Lemma 6.4.

Let $(n_0, \dots, n_L) \stackrel{\text{def}}{=} \vec{n}^{\text{in}}(\mathcal{S})$. We show first that, for every $i \geq 1$, $n_i \leq 2K + T$. Towards a contradiction, let us assume that for some $i \geq 1$, $n_i > 2K + T$.

Let $\alpha' = \beta \alpha'_1 \cdots \alpha'_{n_i}$ be a reordering of $\alpha_1 \cdots \alpha_{N-1}$ such that $\alpha'_1, \dots, \alpha'_{n_i}$ are insertion subsequences that color the n_i elements with color c_i and β contains all other insertions. By minimality of N , all involved states are locally history independent and therefore the reordering does not affect the resulting state, i.e., $\mathcal{P}_{\beta \alpha'_1 \cdots \alpha'_{n_i}}(\mathcal{D}_0) = \mathcal{S}$.

We denote, for every $j \leq n_i$, the state $\mathcal{P}_{\beta \alpha'_1 \cdots \alpha'_j}(\mathcal{D}_0)$ by \mathcal{S}_j and its type function by f_j . We can conclude that $\mathcal{S}_j \simeq_K \mathcal{S}_{j'}$, for all $K \leq j < j' \leq n_i - K - 1$, since

- in \mathcal{S}_K , there are more than $K + T$ uncolored elements and K elements of color c_i ,
- $\alpha'_{K+1} \cdots \alpha'_{n_i-K-1}$ only colors uncolored elements with color c_i , and
- in \mathcal{S}_{n_i-K-1} there are still more than K uncolored elements.

Since there are more than T states between \mathcal{S}_K and \mathcal{S}_{n_i-K-1} , two of them must have the same type function. That is, there must be j_1, j_2 with $K \leq j_1 < j_2 \leq n_i - K - 1$ and $f_{j_1} = f_{j_2}$ and therefore $\mathcal{S}_{j_1} \sim_K \mathcal{S}_{j_2}$.

Let \mathcal{D}'_0 be the empty database resulting from \mathcal{D}_0 by deleting all elements that are colored by the sequence $\alpha'_{j_1+1} \cdots \alpha'_{j_2}$. Since \mathcal{D}'_0 has more than $j_1 + K > K > q$ elements, $\mathcal{S}_{\text{INIT}}(\mathcal{D}'_0) \sim_K \mathcal{S}_{\text{INIT}}(\mathcal{D}_0)$, in particular these two states have the same type functions. By inductive application of Lemma 6.6 it is easy to show that $\mathcal{P}_{\beta \alpha'_1 \cdots \alpha'_{j_1}}(\mathcal{D}'_0) \sim_K \mathcal{P}_{\beta \alpha'_1 \cdots \alpha'_{j_1}}(\mathcal{D}_0)$.

In the inductive step, we start from two corresponding states whose \sim_K -equivalence has already been established. In particular, they agree on all formulas φ_F and therefore the application of the same one element coloring sequence yields for both the same type function, thanks to Lemma 6.6 and because the reached states are homogeneous by minimality of n and N . Since the number of elements for each (proper) color is the same in both new states and both have more than K uncolored elements, they are also equivalent with respect to \simeq_K .

For each j with $j_2 \leq j \leq N - 1$ let $\mathcal{S}'_j \stackrel{\text{def}}{=} \mathcal{P}_{\beta \alpha'_1 \cdots \alpha'_{j_1} \alpha'_{j_2+1} \cdots \alpha'_j}(\mathcal{D}'_0)$.

We emphasize that, for every j , $\vec{n}^{\text{in}}(\mathcal{S}'_j)$ and $\vec{n}^{\text{in}}(\mathcal{S}_j)$ only differ in their entry for color c_i (which for both is at least K). In particular, they have the same number of uncolored elements.

Thus, $\mathcal{S}'_{j_2} \sim_K \mathcal{S}_{j_1} \sim_K \mathcal{S}_{j_2}$ and therefore, as before, \mathcal{S}'_{j_2} and \mathcal{S}_{j_2} agree on all formulas φ_F . It follows that the two states \mathcal{S}'_{j_2+1} and \mathcal{S}_{j_2+1} obtained by the sequence α'_{j_2+1} again have the same type function. As they both have at least K uncolored elements and at least K elements with color c_i (and agree on all other color frequencies), we get $\mathcal{S}'_{j_2+1} \sim_K \mathcal{S}_{j_2+1}$. An inductive application of the same argument yields $\mathcal{S}'_{N-1} \sim_K \mathcal{S}_{N-1} = \mathcal{S}$. Since $\mathcal{S} \models \varphi_{\text{bad}}$ we conclude $\mathcal{S}'_{N-1} \models \varphi_{\text{bad}}$ and thus \mathcal{S}'_{N-1} is a bad state. As \mathcal{S}'_{N-1} can be reached by fewer insertions than \mathcal{S} we get the desired contradiction and thus $n_i \leq 2K + T$, for all $i \geq 1$.

We finally show that $n_0 \leq K$. Otherwise, if $n_0 > K$, we could replace \mathcal{D}_\emptyset by the empty database \mathcal{D}'_\emptyset in which one element that is uncolored in \mathcal{S} is removed. Similarly as before it would follow that $\mathcal{P}_{\alpha_1 \dots \alpha_{N-1}}(\mathcal{D}'_\emptyset) \sim_K \mathcal{P}_{\alpha_1 \dots \alpha_{N-1}}(\mathcal{D}_\emptyset)$ and therefore that $\mathcal{P}_{\alpha_1 \dots \alpha_{N-1}}(\mathcal{D}'_\emptyset)$ satisfies φ_{bad} and is therefore bad, contradicting the choice of \mathcal{D}_\emptyset . This completes the proof of the proposition. \blacktriangleleft

We can now conclude the main result of this section.

► **Theorem 6.8.** *HISTORYINDEPENDENCE is decidable for DYNFO(1-in)-programs.*

Proof. It follows immediately from Proposition 6.7 that Algorithm 1 is a correct decision algorithm for HISTORYINDEPENDENCE of DYNFO(1-in)-programs.

Algorithm 1 Deciding HISTORYINDEPENDENCE for DYNFO(1-in)-programs

Input: A DYNFO(1-in, m -aux)-program \mathcal{P} with ℓ input relations and quantifier depth q .

- 1: Let K , L and T be as in Proposition 6.7.
 - 2: **for** all empty databases \mathcal{D}_\emptyset over domains $\{1, \dots, n\}$ with $n \leq (2K + T)(L + 1)$ **do**
 - 3: **for** all normal form insertion sequences α over $\{1, \dots, n\}$ **do**
 - 4: **if** $\mathcal{P}_\alpha(\mathcal{D}_\emptyset)$ is not homogeneous or not locally history independent **then** Reject.
 - 5: **end for**
 - 6: **end for**
 - 7: Accept.
-

Using the same technique as used in the proof of Theorem 4.7(b), history independence can be shown to be decidable for DYNPROP(1-aux)-programs.

► **Theorem 6.9.** *HISTORYINDEPENDENCE is decidable for DYNPROP(1-aux)-programs.*

Proof. Let \mathcal{P} be a DYNPROP(ℓ -in, 1-aux)-program for some $\ell \in \mathbb{N}$. Recall that, according to Lemma 6.2, for testing history independence it suffices to check that no non-locally history independent state can be reached by an insertion sequence in normal form. We argue that if a non-locally history independent state can be reached by \mathcal{P} , then such a state with few tuples in the input relations can be reached as well. History independence can then be tested in a brute force manner by trying out insertion sequences for all input databases with few tuples.

Suppose that \mathcal{S} is a non-locally history independent state reachable by \mathcal{P} such that the number N of tuples in input databases of \mathcal{S} is minimal. In particular, \mathcal{P} is history independent for input databases with less than N tuples, that is, all modification sequences α and α' yielding an input database with less than N tuples also yield the same state. Let \vec{a} be an 2ℓ -ary tuple that witnesses that \mathcal{S} is not locally history independent, i.e. there are two modifications on \vec{a} that contradict (H1), (H2) or (H3). Further let γ be the atomic type of \vec{a} . Now, using the same argument as in the proof of Theorem 4.7 as well as the history independence of \mathcal{P} for databases with less than N tuples, one can show that for exhibiting a tuple of type γ the number N of input tuples does not have to be large. \blacktriangleleft

7 Conclusion

In this work we studied the algorithmic properties of static analysis problems for (restrictions of) dynamic programs. Most of the results are summarized in Table 1. In general only very strong restrictions yield decidability.

The only cases left open are about DYNPROP-programs when both the arity of the input and the arity of the auxiliary relations is at least 2. For such programs the status of history independence and emptiness of consistent remains open. We conjecture that for history independence the decidable fragment of DYNPROP is larger than exhibited here.

Our results will hopefully contribute to a better understanding of the power of dynamic programs. On the one hand the undecidability proofs show that very restricted dynamic programs can already simulate powerful machine models. It is natural to ask whether this power can be used to maintain other, more common queries. On the other hand the decidability results utilize limitations of the state space and the transition between states for classes of restricted programs. Such limitations can be a good starting point for the development of techniques for proving lower bounds for the respective fragments.

References

- 1 Julius R. Büchi and Calvin C. Elgot. Decision problems of weak second order arithmetics and finite automata, Part I. *Notices of the American Mathematical Society*, 5:834, 1958.
- 2 Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability is in DynFO. In *ICALP*, pages 159–170, 2015.
- 3 Guozhu Dong, Leonid Libkin, and Limsoon Wong. On impossibility of decremental recomputation of recursive queries in relational calculus and SQL. In *DBPL*, page 7, 1995.
- 4 Guozhu Dong, Leonid Libkin, and Limsoon Wong. Incremental recomputation in local languages. *Inf. Comput.*, 181(2):88–98, 2003.
- 5 Guozhu Dong and Jianwen Su. Deterministic FOIES are strictly weaker. *Ann. Math. Artif. Intell.*, 19(1-2):127–146, 1997.
- 6 Guozhu Dong and Jianwen Su. Arity bounds in first-order incremental evaluation and definition of polynomial time database queries. *J. Comput. Syst. Sci.*, 57(3):289–308, 1998.
- 7 Guozhu Dong, Jianwen Su, and Rodney Topor. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14, 1995.
- 8 Guozhu Dong and Limsoon Wong. Some relationships between the FOIES and Σ_1^1 arity hierarchies. *Bulletin of the EATCS*, 61, 1997.
- 9 Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In *ICALP*, pages 103–115, 1998.
- 10 Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Perspectives in Mathematical Logic. Springer, 1995.
- 11 Paul Erdős and Richard Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, s1-35(1):85–90, 1960.
- 12 Wouter Gelade, Marcel Marquardt, and Thomas Schwentick. The dynamic complexity of formal languages. *ACM Trans. Comput. Log.*, 13(3):19, 2012.
- 13 Erich Grädel and Sebastian Siebertz. Dynamic definability. In *ICDT*, pages 236–248, 2012.
- 14 William Hesse. *Dynamic Computational Complexity*. PhD thesis, University of Massachusetts Amherst, 2003.
- 15 John E. Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theor. Comput. Sci.*, 8:135–159, 1979.
- 16 Stasys Jukna. *Extremal combinatorics*, volume 2. Springer, 2001.
- 17 Stefan Kratsch and Magnus Wahlström. Preprocessing of min ones problems: A dichotomy. In *ICALP*, pages 653–665, 2010.
- 18 Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.
- 19 Dániel Marx. Parameterized complexity of constraint satisfaction problems. *Computational Complexity*, 14(2):153–183, 2005.

- 20 Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.
- 21 Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. In *PODS*, pages 210–221. ACM Press, 1994.
- 22 Boris A. Trahtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *AMS Translations, Series 2*, 23:1–5, 1963.
- 23 Nils Vortmeier. Komplexitätstheorie verlaufsunabhängiger dynamischer Programme. Master thesis (in German).
- 24 Thomas Zeume. The dynamic descriptive complexity of k-clique. In *MFCS*, pages 547–558, 2014.
- 25 Thomas Zeume and Thomas Schwentick. Dynamic conjunctive queries. In *ICDT*, pages 38–49, 2014.
- 26 Thomas Zeume and Thomas Schwentick. On the quantifier-free dynamic complexity of reachability. *Inf. Comput.*, 240:108–129, 2015.